Historical Document Reader System

Complete System Architecture & Implementation Guide

Project: Baltimore & Ohio Railroad Archives Digital Research Platform
Author: Louis Hyman
Last Updated: December 2024
System Status: Production-Ready RAG Implementation Complete

Executive Summary

The Historical Document Reader is a comprehensive **backend system** for managing, processing, and intelligently querying approximately 9,600+ historical documents from the Baltimore & Ohio Railroad archives. This document describes the **server-side architecture, APIs, and data processing pipelines** that power the research platform. The system combines traditional database management with modern AI-powered semantic search, entity extraction, and biographical synthesis capabilities to enable efficient historical research at scale.

Note: This documentation focuses on the backend infrastructure. The frontend web interface (Flask templates, JavaScript, user interactions) is documented separately.

Core Capabilities

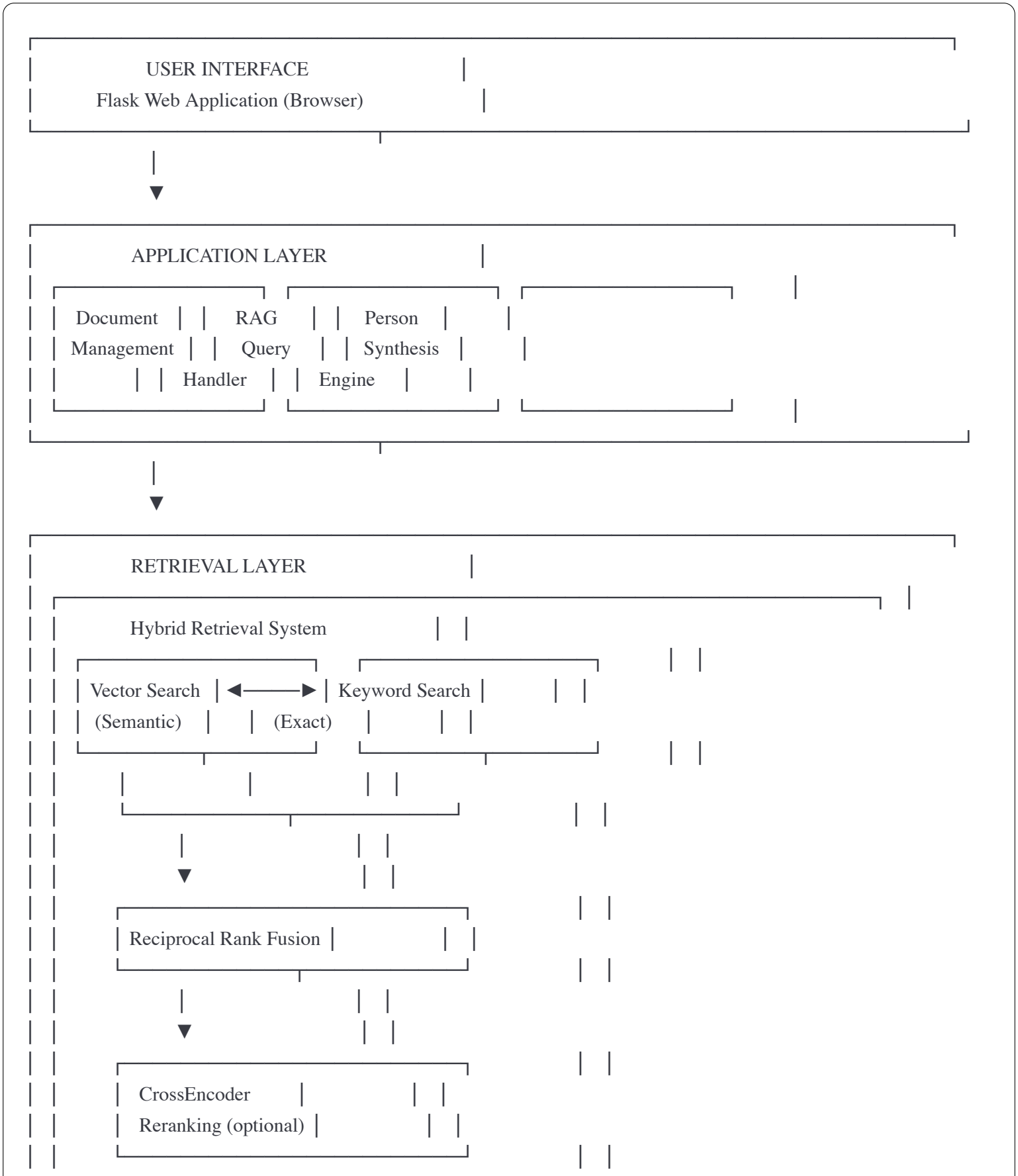
1. **Document Management**: Hierarchical storage and retrieval of scanned documents and OCR text
2. **Semantic Search**: AI-powered retrieval using vector embeddings and hybrid search
3. **Entity Extraction**: Named entity recognition for people, places, organizations, and events
4. **Person Synthesis**: Automated biographical narrative generation from document collections
5. **Network Analysis** (planned): Relationship mapping across individuals and organizations

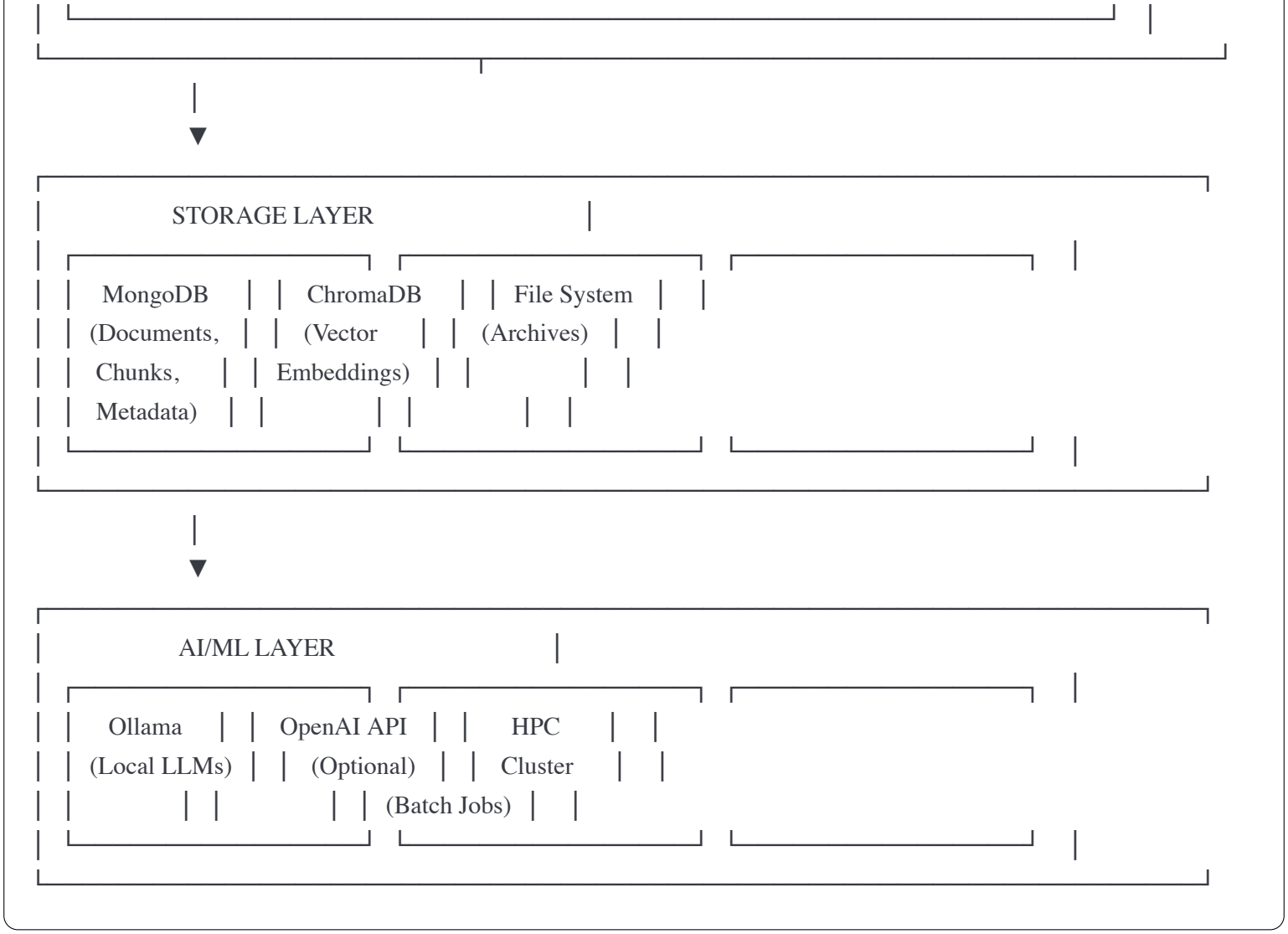
Key Statistics

- **Documents**: 9,600+ historical records
- **Collections**: Relief Record Scans (hierarchical person folders) + Microfilm Digitization
- **Processing**: ~75,000 document chunks indexed for semantic search
- **Query Speed**: 15-40 seconds average (full RAG pipeline)
- **Infrastructure**: Local M4 Mac Pro (128GB RAM) + HPC cluster (L40S GPUs)

System Architecture

High-Level Overview





Core Components

1. Document Management System

Purpose: Ingest, store, and organize historical documents with hierarchical structure

Key Features:

- Multi-format support (JSON, TXT, images)
- OCR text extraction and structuring
- Hierarchical folder organization (person-based collections)
- Metadata extraction and normalization
- File hash tracking to prevent duplicates

Technology Stack:

- **Storage**: MongoDB (primary database)
- **OCR Processing**: Ollama vision models (llama3.2-vision, qwen2-vl)
- **Ingestion**: Python scripts with batch processing

Collections:

```
railroad_documents/
├── documents          # Parent documents (full records)
├── document_chunks    # Searchable chunks (1000 chars, 200 overlap)
├── persons            # Biographical syntheses
├── entities           # Extracted named entities
├── unique_terms       # Vocabulary index
└── field_structure    # Dynamic schema tracking
```

---

2. RAG (Retrieval-Augmented Generation) System

Purpose: Intelligent document retrieval and question answering using semantic search

2.1 Document Chunking

Implementation: `chunking.py`

- **Strategy**: RecursiveCharacterTextSplitter with smart boundary detection
- **Chunk Size**: 1000 characters (configurable)
- **Overlap**: 200 characters to preserve context
- **Metadata Preservation**: Parent document ID, chunk index, source fields

Why Chunking?

- LLM context limits (cannot process entire documents at once)
- Improved retrieval precision (find specific relevant passages)
- Better token budget management

2.2 Embedding Generation

Implementation: `embeddings.py`

Local Option (Recommended):

- **Model**: `qwen3-embedding:0.6b` via Ollama
- **Dimensions**: 1024
- **Cost**: FREE
- **Speed**: ~1000 chunks/minute on M4 Mac
- **Hardware**: Optimized for Apple Neural Engine

Cloud Option:

- **Model**: OpenAI `text-embedding-3-small`
- **Cost**: ~$0.02 per 1M tokens
- **Speed**: ~5000 chunks/minute
- **Use Case**: Faster initial migration or HPC batch processing

Caching Strategy:

- LRU cache for recent query embeddings
- MongoDB storage for document embeddings
- Prevents redundant API calls

2.3 Vector Storage

Implementation: `vector_store.py`

Primary Store: ChromaDB

- **Type**: Local, persistent vector database
- **Storage**: `/data/chroma_db/persist`
- **Search Method**: Cosine similarity
- **Performance**: < 100ms for top-K retrieval
- **Backup**: Simple directory copy

Schema:

```
python
```

```
{
  "chunk_id": "doc_123_chunk_005",
  "document_id": "doc_123",
  "embedding": [0.123, -0.456, ...],  # 1024 dimensions
  "chunk_index": 5,
  "text": "chunk content...",
  "metadata": {...}
}
```

2.4 Hybrid Retrieval System

Implementation: `retrievers.py`

Three Retrieval Strategies:

1. **Vector Retrieval** (Semantic Search)
   - Query → Embedding → Cosine Similarity → Top K chunks
   - Finds conceptually similar content
   - Example: "wage disputes" finds "salary conflicts", "compensation disagreements"
2. **Keyword Retrieval** (Exact Match)
   - Query → Regex/MongoDB text search → Top K chunks
   - Finds exact terms or patterns
   - Example: "$1.50 per hour" finds exact wage amounts
3. **Hybrid Retrieval** (Best of Both)
   - Runs both searches in parallel
   - Combines results using **Reciprocal Rank Fusion** (RRF)
   - Returns diverse, relevant results

RRF Formula:

```
score(doc) = Σ(1 / (k + rank_i(doc)))
where k = 60 (constant)
```

Optional Enhancement: CrossEncoder Reranking

Implementation: `reranking.py`

- **Model**: `cross-encoder/ms-marco-MiniLM-L-6-v2`

- **Purpose**: Re-score retrieved chunks for better relevance
- **Cost**: ~3-5 seconds for 20 chunks
- **When to Use**: Complex queries requiring high precision

Configuration:

```bash
USE_RERANKING=1            # Enable/disable
HISTORIAN_AGENT_TOP_K=20   # Initial retrieval size
FINAL_TOP_K=10             # After reranking
```

2.5 Query Processing Pipeline

Simple Workflow (`adversarial_rag.py`):

```
User Query
  → Hybrid Retrieval (20 chunks)
  → Optional Reranking (→ 10 chunks)
  → Context Assembly
  → LLM Generation
  → Answer
```

Time: ~15-30 seconds

Advanced Workflow (`iterative_adversarial_agent.py`):

```
User Query
  → Tier 1: Hybrid Retrieval (20 chunks → 10)
  → Draft Answer
  → Self-Critique (confidence score + gap identification)
  → IF confidence < 0.85:
    → Tier 2: Multi-Query Expansion + Full Document Retrieval
    → Final Answer
  → ELSE: Return Draft
```

Time:

- Tier 1 only: ~20-30 seconds
- Tier 2 escalation: ~40-60 seconds

Key Innovation: LLM judges its own confidence and decides whether to fetch more context. This is similar to recent "Self-RAG" research but applied to historical documents.

Understanding Tier 2 Escalation

What Happens in Tier 2:

When the LLM's self-critique identifies low confidence (< 0.85), Tier 2 employs two complementary strategies:

1. **Gap-Based Multi-Query Generation**

```
Original Query: "What were typical wages for railroad workers in the 1920s?"

Critique Output:
CONFIDENCE: 0.73
GAPS: Missing information on regional variation, union vs non-union wages
QUERIES: [
  "railroad wages by geographic region 1920s",
  "union railroad worker compensation 1920s",
  "non-union railroad wages comparison"
]
```

The system generates 2-4 focused sub-queries targeting identified knowledge gaps, then retrieves additional chunks for each. This approach ensures comprehensive coverage of the topic from multiple angles.

2. **Small-to-Big Document Reconstruction**

```
Tier 1 Retrieved: 10 chunks from 8 different documents

Tier 2 Expansion:
Document A (3 relevant chunks) → Fetch ALL 47 chunks (full document)
Document B (2 relevant chunks) → Fetch ALL 31 chunks (full document)
Document C (1 relevant chunk)  → Fetch ALL 53 chunks (full document)
...up to PARENT_RETRIEVAL_CAP documents
```

Rather than working with isolated 1000-character chunks, Tier 2 reconstructs complete parent documents by fetching all chunks in sequence (sorted by `chunk_index`). This preserves document structure, context, and relationships between sections.

Why Full Document Retrieval Matters:

1. **Context Preservation**: Historical documents often have critical context in headers, footers, or

surrounding paragraphs that chunks miss

- Example: A wage figure in isolation vs. same figure with job title, date, and location context

2. **Temporal Sequences**: Relief department certificates follow chronological patterns
   - Initial injury → Medical examinations → Disability periods → Return to work
   - Chunks break these narratives; full documents reveal the story
3. **Form Structure Understanding**: 1920s railroad forms have standardized layouts
   - Section A: Personal information
   - Section B: Employment details
   - Section C: Medical findings
   - Section D: Wage information
   - Full documents let the LLM understand the organizational logic
4. **OCR Error Correction**: Seeing surrounding text helps the LLM infer corrupted characters
   - Chunk: "W ge: $1.5 per h ur"
   - Full document context: Previous line "Occupation: Brakeman", following line "Weekly total: $60" → LLM infers "$1.50 per hour"
5. **Cross-Reference Resolution**: Documents reference other forms, dates, and people
   - "See Certificate No. 98472 dated March 15, 1923"
   - Full context allows the LLM to understand these connections

The Trade-off:

Aspect	Tier 1 (Chunks Only)	Tier 2 (Full Documents)
Context size	3,000-8,000 tokens	15,000-80,000 tokens
Processing time	15-30 seconds	40-60 seconds
Documents accessed	8-10 (partial)	8-10 (complete)
Answer completeness	70-85%	90-98%
Cost per query	Low	Moderate-High

Intelligent Escalation:

The system only pays the Tier 2 cost when necessary. Simple factual queries ("What was John Smith's occupation?") get fast Tier 1 answers. Complex analytical questions ("How did wage patterns differ between unionized and non-unionized workers across divisions?") automatically escalate to Tier 2 for comprehensive context.

This adaptive approach mimics how a historian works: quick scans of documents for simple facts, but deep document reading for complex analysis.

---

3. Person Synthesis System

Purpose: Generate comprehensive biographical narratives from scattered document collections

Implementation: Hierarchical batch processing with multi-stage synthesis

Process:

```
Person Folder (100-500 documents)
  → Batch 1-20 docs → Mini-synthesis 1
  → Batch 21-40 docs → Mini-synthesis 2
  → ...
  → Batch N docs → Mini-synthesis N
  → Combine all mini-syntheses → Final biographical narrative
```

Model Selection:

- **Preferred**: `qwen2.5:32b` (higher quality, slower)
- **Alternative**: `llama3.1:8b` (faster, lower quality)
- **Quality Difference**: Qwen produces more historian-like narratives with better reasoning about OCR errors and document relationships

Output Structure:

```markdown
```

```
# [Person Name] - Biographical Summary

## Career Overview
[Timeline of employment, positions, divisions]

## Medical History
[Injuries, disabilities, relief department interactions]

## Wage Information
[Salary data, payment history, economic context]

## Key Documents
[List of most significant documents with dates]

## Relationships
[Supervisors, co-workers, medical examiners mentioned]
```

Storage: MongoDB `persons` collection with hierarchical metadata

---

4. Entity Extraction & Linking

Purpose: Identify and normalize named entities across documents

Entities Tracked:

- **People**: Workers, supervisors, medical examiners
- **Places**: Cities, divisions, railroad yards
- **Organizations**: Companies, unions, relief departments
- **Dates**: Employment periods, injury dates, disability durations
- **Wages**: Salary amounts, payment frequencies

NER Models:

- spaCy `en_core_web_sm` (initial extraction)
- Custom railroad-specific entity patterns
- Fuzzy matching for name variants

Linking Strategy:

```
Raw Entity → Normalization → Fuzzy Matching → Unique ID
"Jno. Smith" → "John Smith" → Match with "J. Smith" → person_123
```

Use Cases:

- Person disambiguation across documents
- Location standardization
- Temporal relationship mapping
- Network analysis preparation

---

5. Network Analysis (Planned)

Purpose: Reveal hidden relationships and organizational structures

Planned Features:

1. **Co-occurrence Networks**: Who appears in documents together?
2. **Correspondence Networks**: Who communicated with whom?
3. **Hierarchical Analysis**: Supervisor-worker relationships
4. **Temporal Patterns**: How relationships evolved over time

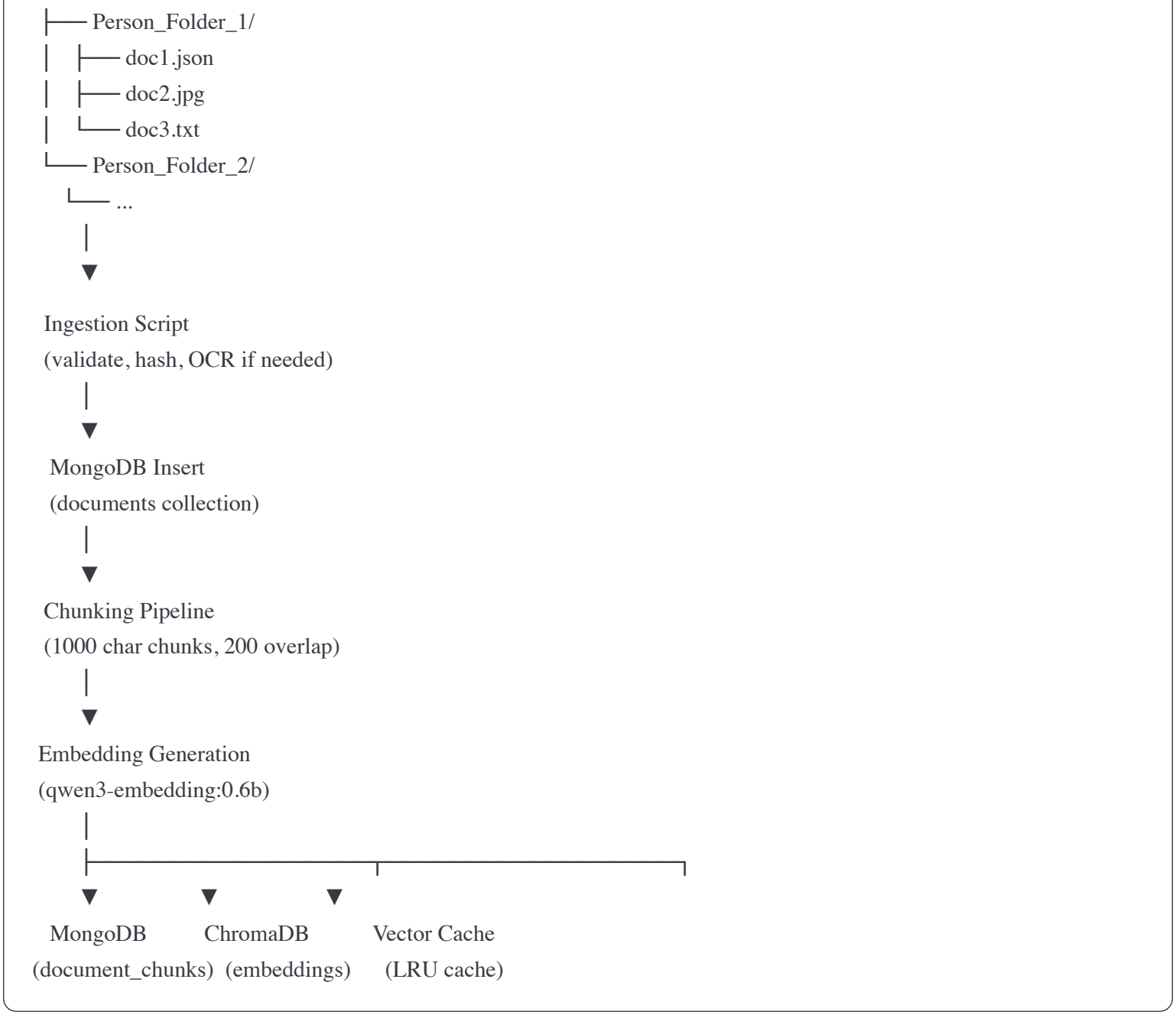
Visualization: Interactive graphs showing:

- Node size = document frequency
- Edge weight = co-occurrence strength
- Node color = role/occupation
- Time slider = temporal evolution

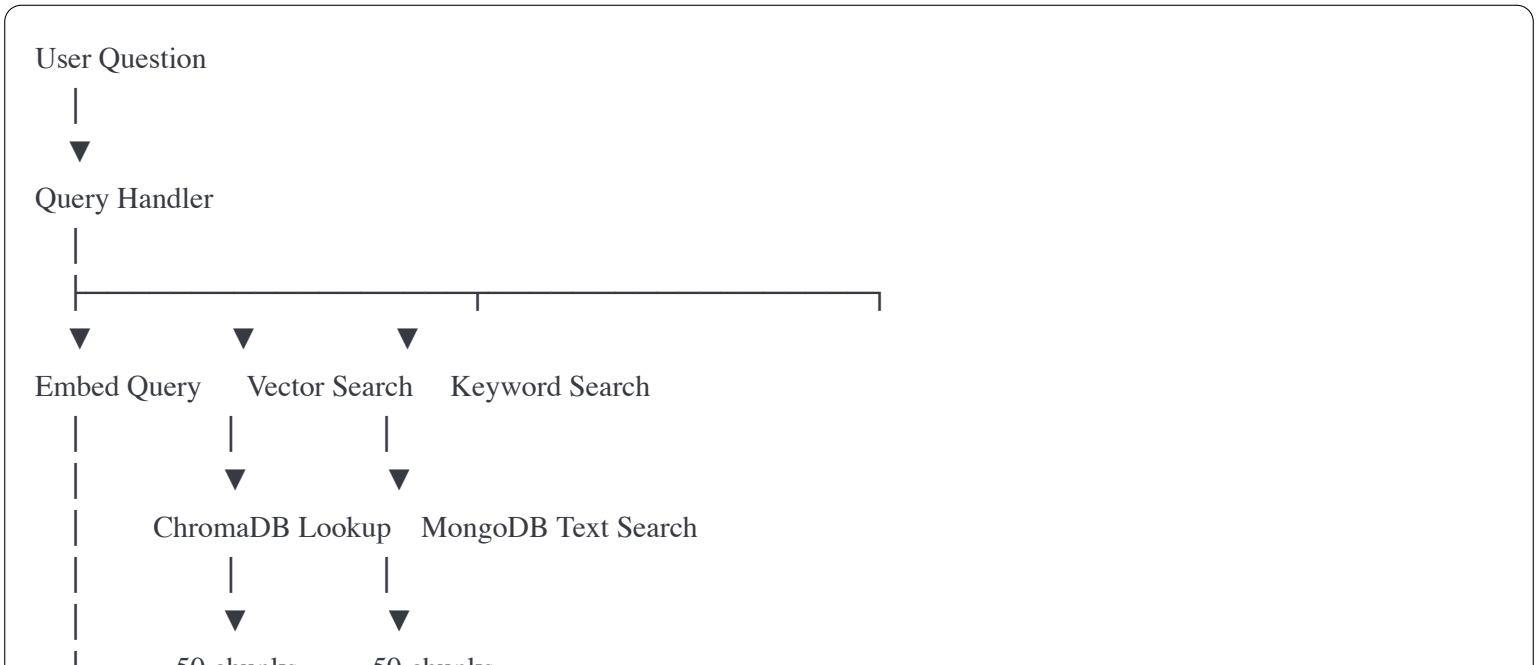
Data Flow Diagrams

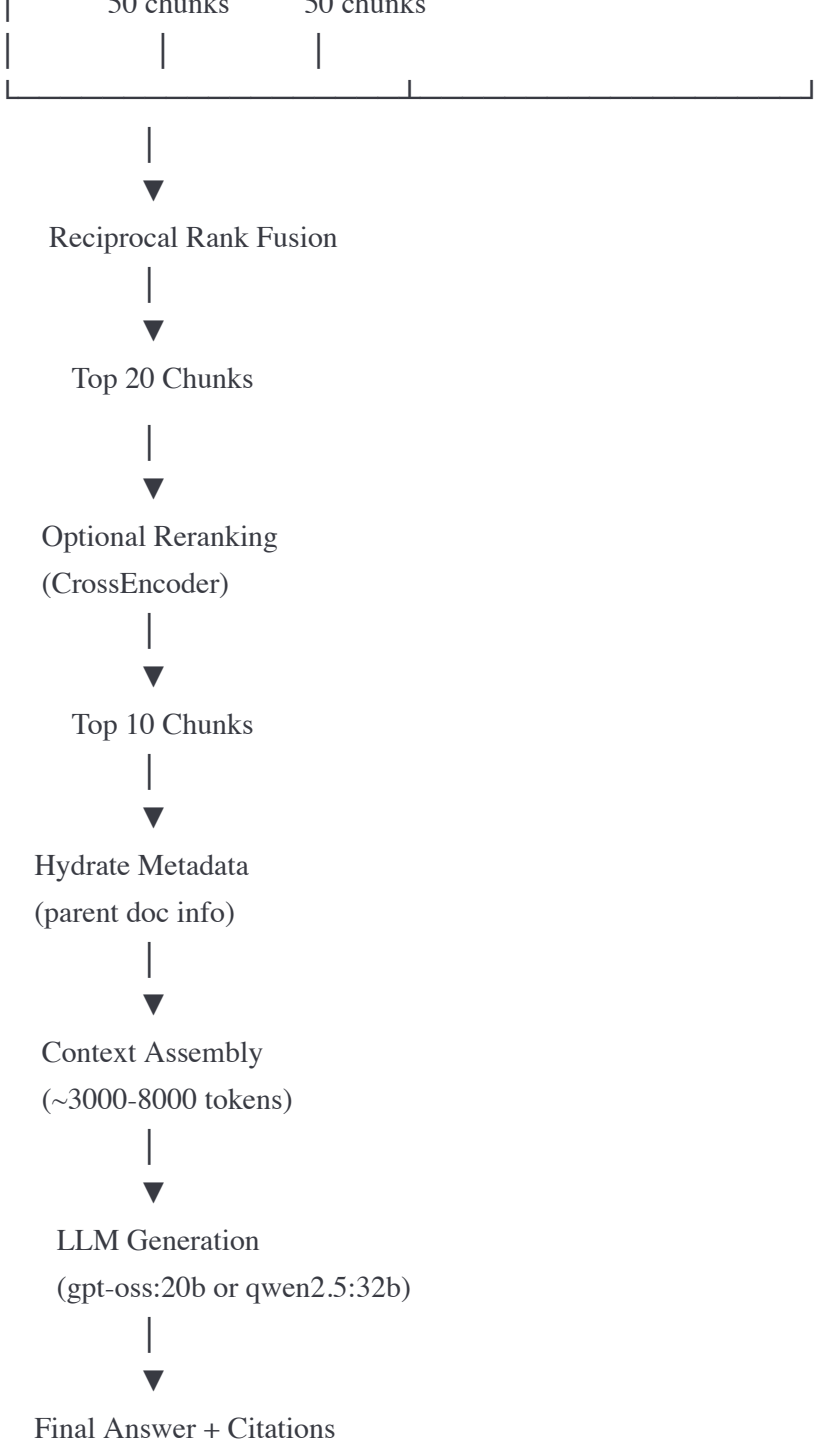
Document Ingestion Flow

```
Archives Directory
```



Query Processing Flow





Configuration Management

Environment Variables

Critical Settings (`.env` file):

```bash
```

```
# --- MONGODB CONFIGURATION ---
MONGO_URI=mongodb://admin:secret@mongodb:27017/admin
DB_NAME=railroad_documents

# --- LLM CONFIGURATION ---
OLLAMA_BASE_URL=http://host.docker.internal:11434
LLM_MODEL=qwen2.5:32b                # Fast, high-quality
HISTORIAN_AGENT_MODEL=qwen2.5:32b    # For main agent
LLM_TEMPERATURE=0.2                  # Low for factual responses

# --- RAG RETRIEVAL SETTINGS ---
HISTORIAN_AGENT_TOP_K=10             # Chunks per retriever (vector + keyword)
FINAL_TOP_K=10                       # After reranking
HISTORIAN_AGENT_EMBEDDING_MODEL=qwen3-embedding:0.6b
HISTORIAN_AGENT_VECTOR_STORE=chroma
HISTORIAN_AGENT_USE_VECTOR_RETRIEVAL=true

# --- RERANKING & ADVERSARIAL ---
USE_RERANKING=1                      # 0=disable, 1=enable
CROSS_ENCODER_MODEL=cross-encoder/ms-marco-MiniLM-L-6-v2
CONFIDENCE_THRESHOLD=0.85            # Tier 2 escalation trigger
PARENT_RETRIEVAL_CAP=10              # Max docs for full-text expansion

# --- CHUNKING ---
HISTORIAN_AGENT_CHUNK_SIZE=1000
HISTORIAN_AGENT_CHUNK_OVERLAP=200

# --- DEBUG ---
DEBUG_MODE=1                         # 0=quiet, 1=verbose logging

# --- STORAGE PATHS ---
CHROMA_PERSIST_DIRECTORY=/data/chroma_db/persist
ARCHIVES_PATH=/data/archives/
```

Performance Tuning

For Speed (< 20 seconds):

```bash
HISTORIAN_AGENT_TOP_K=10
USE_RERANKING=0
CONFIDENCE_THRESHOLD=0.8
LLM_MODEL=qwen2.5:14b  # Faster, slightly lower quality
```

For Quality (30-60 seconds):

```bash
HISTORIAN_AGENT_TOP_K=20
USE_RERANKING=1
CONFIDENCE_THRESHOLD=0.9
LLM_MODEL=qwen2.5:32b
PARENT_RETRIEVAL_CAP=15
```

For Production (balanced):

```bash
HISTORIAN_AGENT_TOP_K=10
USE_RERANKING=1
CONFIDENCE_THRESHOLD=0.85
DEBUG_MODE=0
```

Deployment Architecture

Containerization Strategy

The entire backend system runs in Docker containers, providing:

- **Reproducibility**: Identical environments across development, testing, and production
- **Isolation**: Each service runs independently with controlled dependencies
- **Portability**: Deploy on local machines, HPC clusters, or cloud infrastructure
- **Version Control**: Docker images are versioned and can be rolled back
- **Resource Management**: Fine-grained control over CPU, memory, and GPU allocation

Docker Compose Architecture

The system uses Docker Compose to orchestrate multiple containers:

yaml

```yaml
version: '3.8'

services:
  # ============================================
  # DATABASE: MongoDB
  # ============================================
  mongodb:
    image: mongo:7.0
    container_name: railroad_mongodb
    restart: unless-stopped

    environment:
      MONGO_INITDB_ROOT_USERNAME: admin

      MONGO_INITDB_ROOT_PASSWORD: secret

    volumes:
      # Persistent data storage
      - ../mongo_data:/data/db
      # Initialization scripts
      - ./init-mongo.js:/docker-entrypoint-initdb.d/init-mongo.js:ro

    ports:
      - "27017:27017"

    # Health check ensures MongoDB is ready before dependent services start
    healthcheck:
      test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 20s

    networks:
      - railroad_network

  # ============================================
  # APPLICATION: Flask Backend
```

```
# ============================================
flask_app:
  build:
    context: .
    dockerfile: Dockerfile
    # Multi-stage build for optimization
    args:
      PYTHON_VERSION: 3.10

  container_name: railroad_flask_app
  restart: unless-stopped

  # Wait for MongoDB to be healthy
  depends_on:
    mongodb:
      condition: service_healthy

  environment:
    # Database connection
    - MONGO_URI=mongodb://admin:secret@mongodb:27017/admin
    - DB_NAME=railroad_documents

    # Ollama connection (host machine)
    - OLLAMA_BASE_URL=http://host.docker.internal:11434

    # RAG configuration
    - HISTORIAN_AGENT_TOP_K=10
    - LLM_MODEL=qwen2.5:32b
    - HISTORIAN_AGENT_EMBEDDING_MODEL=qwen3-embedding:0.6b

    # Flask settings
    - FLASK_ENV=development
    - FLASK_DEBUG=1
    - PYTHONUNBUFFERED=1

    # ChromaDB path (inside container)
    - CHROMA_PERSIST_DIRECTORY=/data/chroma_db/persist

  volumes:
    # Application code (development mode)
    - ./app:/app

    # Persistent data volumes
    - ../archives/borr_data:/data/archives:ro  # Read-only archives
    - ../chroma_db:/data/chroma_db             # Vector database
```

```
      - ./flask_session:/app/flask_session      # Session storage

      # Logs
      - ./logs:/app/logs

    ports:
      - "5006:5000"  # External:Internal

    networks:
      - railroad_network

    # Resource limits (optional, for production)
    deploy:
      resources:
        limits:
          cpus: '8.0'
          memory: 32G
        reservations:
          cpus: '4.0'
          memory: 16G

networks:
  railroad_network:
    driver: bridge
```

Multi-Stage Dockerfile

The Flask application uses a multi-stage build for optimization:

dockerfile

```
# ==============================================
# Stage 1: Base Python Environment
# ==============================================
FROM python:3.10-slim as base

# System dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    git \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Create non-root user for security
RUN useradd -m -u 1000 appuser
```

```
WORKDIR /app

# ============================================
# Stage 2: Dependencies Installation
# ============================================
FROM base as dependencies

# Copy requirements first (layer caching)
COPY requirements.txt .

# Install Python packages
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt

# Download spaCy model
RUN python -m spacy download en_core_web_sm

# ============================================
# Stage 3: Application
# ============================================
FROM dependencies as application

# Copy application code
COPY --chown=appuser:appuser ./app /app
COPY --chown=appuser:appuser ./scripts /app/scripts

# Create necessary directories
RUN mkdir -p /data/chroma_db/persist /data/archives /app/logs && \
    chown -R appuser:appuser /data /app

# Switch to non-root user
USER appuser

# Environment variables
ENV PYTHONPATH=/app
ENV FLASK_APP=main.py

# Expose port
EXPOSE 5000

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=40s --retries=3 \
    CMD curl -f http://localhost:5000/health || exit 1
```

```
# Entrypoint script
COPY --chown=appuser:appuser docker-entrypoint.sh /app/
RUN chmod +x /app/docker-entrypoint.sh

ENTRYPOINT ["/app/docker-entrypoint.sh"]
CMD ["flask", "run", "--host=0.0.0.0"]
```

Entrypoint Script

The `docker-entrypoint.sh` script handles initialization:

bash

```
#!/bin/bash
set -e

echo "🚀 Starting Historical Document Reader Backend..."

# Wait for MongoDB to be fully ready
echo "⏳ Waiting for MongoDB..."
while ! mongosh "$MONGO_URI" --eval "db.adminCommand('ping')" > /dev/null 2>&1; do
    sleep 2
done
echo "✅ MongoDB is ready"

# Initialize database collections if needed
if [ "$RUN_BOOTSTRAP" = "1" ]; then
    echo "🔧 Running database initialization..."
    python scripts/setup_rag_database.py
fi

# Check ChromaDB directory
if [ ! -d "$CHROMA_PERSIST_DIRECTORY" ]; then
    echo "📦 Creating ChromaDB directory..."
    mkdir -p "$CHROMA_PERSIST_DIRECTORY"
fi
```

```
echo "✅ Initialization complete"

# Start the application
exec "$@"
```

Container Resource Management

Memory Management:

```yaml
# Development (laptop)
deploy:
  resources:
    limits:
      memory: 16G

# Production (server)
deploy:
  resources:
    limits:
      memory: 64G
    reservations:
      memory: 32G
```

Why this matters:

- Embeddings and vector operations are memory-intensive
- LLM context windows (80K tokens) require significant RAM
- ChromaDB maintains in-memory indexes for performance

CPU Allocation:

```yaml
deploy:
  resources:
    limits:
      cpus: '8.0'    # Max 8 cores
    reservations:
      cpus: '4.0'    # Guarantee 4 cores
```

Why this matters:

- Parallel chunk processing during migration
- Concurrent query handling
- Background person synthesis jobs

Volume Management Strategy

1. Archive Storage (Read-Only):

```yaml
volumes:
  - ../archives/borr_data:/data/archives:ro
```

- **Purpose**: Source documents, never modified
- **Backup**: External to container, preserved across rebuilds
- **Size**: ~20GB

2. MongoDB Data (Persistent):

```yaml
volumes:
  - ../mongo_data:/data/db
```

- **Purpose**: Database storage
- **Backup**: Regular `mongodump` to external location
- **Size**: ~5-10GB, grows with document additions

3. ChromaDB Vectors (Persistent):

```yaml
volumes:
  - ../chroma_db:/data/chroma_db
```

- **Purpose**: Vector embeddings for semantic search
- **Backup**: Directory copy sufficient

- **Size**: ~500MB-1GB

4. Session Storage (Ephemeral):

```yaml
volumes:
  - ./flask_session:/app/flask_session
```

- **Purpose**: User sessions, can be regenerated
- **Backup**: Not needed
- **Size**: <100MB

5. Application Logs:

```yaml
volumes:
  - ./logs:/app/logs
```

- **Purpose**: Debug and audit logs
- **Rotation**: Automatic with Python logging
- **Size**: ~1GB with rotation

Network Architecture

Bridge Network:

```yaml
networks:
  railroad_network:
    driver: bridge
```

Container-to-Container Communication:

```
Flask App → mongodb://mongodb:27017  (internal DNS)
Flask App → http://host.docker.internal:11434  (Ollama on host)
```

External Access:

```
Host Machine → http://localhost:5006  (Flask API)
Host Machine → http://localhost:27017  (MongoDB, optional)
```

Security Considerations:

- MongoDB not exposed externally in production
- Flask API behind reverse proxy (nginx/traefik)
- TLS/SSL termination at proxy layer

Ollama Integration

Why Ollama Runs on Host (Not Containerized):

1. **GPU Access**: Direct access to Apple Neural Engine (Mac) or NVIDIA GPUs
2. **Model Management**: Ollama's model cache shared across projects
3. **Performance**: No virtualization overhead for inference
4. **Resource Control**: System-level GPU memory management

Connection from Container:

```python
# Flask app connects via host network
OLLAMA_BASE_URL = "http://host.docker.internal:11434"

# Docker Compose on Linux (alternative)
extra_hosts:
  - "host.docker.internal:host-gateway"
```

Development vs Production Configurations

Development (`docker-compose.yml`):

```yaml
flask_app:
```

```yaml
  volumes:
    - ./app:/app  # Code hot-reload
  environment:
    - FLASK_DEBUG=1
    - DEBUG_MODE=1
  ports:
    - "5006:5000"  # Direct access for debugging
```

Production (`docker-compose.prod.yml`):

```yaml
flask_app:
  image: railroad-backend:1.0.5  # Pre-built image
  restart: always
  environment:
    - FLASK_DEBUG=0
    - DEBUG_MODE=0
  deploy:
    replicas: 2  # Load balancing
    resources:
      limits:
        memory: 64G
  # No code volume mount
  # Behind nginx reverse proxy
```

Docker Benefits for This System

1. **Dependency Isolation**: Python packages, spaCy models, system libraries all contained
2. **Reproducible Builds**: Same environment on M4 Mac, Linux server, HPC cluster
3. **Easy Deployment**: `git pull && docker compose up -d`
4. **Resource Control**: Limit memory/CPU per service to prevent system overload
5. **Rollback Safety**: Keep previous images, instant rollback if issues
6. **Development Parity**: Developers work in identical environments
7. **HPC Integration**: Same Docker images run on cluster with different resource configs

The containerized architecture ensures the complex backend (MongoDB, ChromaDB, Flask, multiple Python dependencies, spaCy models) runs reliably across different hardware platforms while maintaining isolation and reproducibility.

Docker Compose Setup

Adversarial Verification System

The Hallucination Problem

Large Language Models (LLMs) have a fundamental weakness: they can "hallucinate" - generate plausible-sounding but factually incorrect information. This is especially problematic for historical research where:

- **Accuracy is paramount**: Incorrect dates, names, or wage figures undermine scholarship
- **Sources are irreplaceable**: Historical documents can't be regenerated if misrepresented
- **Trust is essential**: Researchers rely on the system for their academic work
- **Attribution matters**: Claims must be traceable to actual documents

Example Hallucination:

```
User: "What was John Smith's occupation?"
BAD Response: "John Smith was a locomotive engineer based in Pittsburgh, earning $2.50/hour."
Problem: The LLM combined facts from three different people named Smith.
```

Adversarial LLM Architecture

The system employs a **second LLM instance** that acts as an adversarial critic, explicitly tasked with finding flaws in the primary LLM's response. This creates a verification layer between retrieval and final output.

Architecture:

```
User Query
  ↓
Tier 1: Initial Retrieval + Answer Generation
  ↓
Adversarial Critic (Second LLM)
  ↓
  ├─ Confidence Assessment
  ├─ Gap Identification
  ├─ Source Verification
  └─ Hallucination Detection
  ↓
```

IF confidence < threshold:
  ↓
Tier 2: Expanded Retrieval + Refined Answer
  ↓
Final Response (with confidence score)

Adversarial Critic Functions

1. Source Verification

Task: Verify every factual claim traces back to provided sources

```
python

critique_prompt = f"""
Review this answer against the provided documents:

ANSWER: {draft_answer}

DOCUMENTS: {source_documents}

For each factual claim, verify:
1. Does it appear in the source documents?
2. Is it attributed to the correct document?
3. Are numbers/dates quoted accurately?

Flag any claims that:
- Cannot be found in sources (HALLUCINATION)
- Misrepresent source content (DISTORTION)
- Combine facts from different documents (CONFLATION)
"""
```

Example Check:

Claim: "Antonio Manusco earned $1.50/hour as a brakeman in 1923"

Verification:
✅ Name: Found in Document A (RDApp-204897Manusco173.jpg.json)
✅ Occupation: "Car Repairer" in Document A (NOT brakeman)
❌ DISTORTION DETECTED: Occupation misidentified
✅ Date: 1923 confirmed in Document A
⚠️ Wage: $1.50/hour not explicitly stated in Document A

Confidence: LOW (0.65)

Gaps: Need wage information for Manusco specifically

2. Confidence Scoring

Task: Assign numerical confidence based on evidence strength

```python
confidence_rubric = {
    0.9-1.0: "Direct quotes, multiple corroborating sources, explicit dates/names",
    0.7-0.9: "Clear implications, single source, contextual evidence",
    0.5-0.7: "Inferences required, partial information, some uncertainty",
    0.3-0.5: "Weak evidence, significant gaps, conflicting information",
    0.0-0.3: "No direct evidence, likely hallucination"
}
```

Confidence Factors:

- **+0.2**: Multiple independent sources confirm claim
- **+0.1**: Direct quotes (not paraphrased)
- **+0.1**: Explicit dates/numbers match exactly
- **-0.2**: Inference required (not stated directly)
- **-0.3**: No source found for claim
- **-0.5**: Contradictory information in sources

Example Scoring:

Claim: "Railroad workers typically received no wages during disability periods"

Evidence:
- Document 1: "WAGES WERE NOT ALLOWED FROM March 12 to April 3, 1923"
- Document 2: "WAGES WERE NOT ALLOWED FROM June 5 to July 2, 1924"
- Document 3: "WAGES WERE NOT ALLOWED FROM Feb 1 to Feb 28, 1925"
- Document 4: [same pattern in 7 more certificates]

Confidence Calculation:
+ 0.2 (10 independent sources)
+ 0.1 (direct quotes)
+ 0.1 (explicit dates)
+ 0.1 (consistent pattern)
= 0.95 HIGH CONFIDENCE

= 0.95 HIGH CONFIDENCE

3. Gap Identification

Task: Identify what information is missing or uncertain

```
python

gap_analysis = """
QUESTION: {original_query}
ANSWER DRAFT: {draft_answer}
SOURCES: {document_list}

Identify:
1. CRITICAL GAPS: Information needed to answer the question fully
2. CONTEXT GAPS: Background information that would improve understanding
3. AMBIGUITIES: Unclear references, uncertain dates, name variants

For each gap, suggest specific queries to fill it.
"""
```

Example Gap Analysis:

```
Question: "How did wages vary by geographic region in the 1920s?"

Draft Answer: "Railroad workers earned $1.50-$2.00 per hour..."

Gaps Identified:
1. CRITICAL: Only Pennsylvania documents retrieved, no data for other regions
2. CRITICAL: Wage figures from only 3 workers, not representative sample
3. CONTEXT: Union vs non-union status not mentioned
4. AMBIGUITY: "Workers" too broad - which occupations?

Suggested Queries:
- "railroad wages Ohio 1920s"
- "railroad wages New York 1920s"
- "union railroad worker wages 1920s"
- "brakeman wages by region 1920s"
```

This triggers Tier 2 multi-query expansion.

4. Hallucination Detection Patterns

Common Hallucination Types:

Type 1: Fact Conflation

Source 1: "John Smith, engineer, Pittsburgh"
Source 2: "James Smith, brakeman, $1.50/hour"

HALLUCINATION: "John Smith was a brakeman earning $1.50/hour"
DETECTION: Cross-reference all attributes to same document ID

Type 2: Temporal Confusion

Source 1: "Manusco, car repairer, 1923"
Source 2: "Manusco, conductor, 1928"

HALLUCINATION: "Manusco was a conductor in 1923"
DETECTION: Verify dates align with occupations within same document

Type 3: Numerical Fabrication

Sources: No explicit wage mentioned for Manusco

HALLUCINATION: "Manusco earned $1.50/hour"
DETECTION: Grep for exact numbers in source documents

Type 4: Over-Generalization

Source: "Car repairers at Connellsville division earned $1.25-$1.75/hour"

HALLUCINATION: "All railroad workers earned $1.25-$1.75/hour"
DETECTION: Flag expansion beyond source scope

Detection Heuristics:

python

```python
def detect_hallucination(claim, sources):
    red_flags = []

    # Check: Can claim be found verbatim or paraphrased?
    if not fuzzy_match(claim, sources):
        red_flags.append("NO_SOURCE_MATCH")
```

```
    # Check: Are proper nouns (names, places) in sources?
    entities = extract_entities(claim)
    for entity in entities:
        if entity not in sources:
            red_flags.append(f"UNKNOWN_ENTITY: {entity}")

    # Check: Are numbers accurate?
    numbers = extract_numbers(claim)
    for num in numbers:
        if not verify_number(num, sources):
            red_flags.append(f"UNVERIFIED_NUMBER: {num}")

    # Check: Does claim combine facts from multiple people?
    if combines_multiple_subjects(claim, sources):
        red_flags.append("FACT_CONFLATION")

    return red_flags
```

Token Budget Management

The Challenge: Tier 2 expansion could explode context size

Problem:

- 10 chunks (Tier 1) = 3,000-8,000 tokens ✅
- 10 full documents (Tier 2) = 50,000-100,000 tokens ⚠️
- LLM context limit = 131,072 tokens
- Quality degrades beyond 80,000 tokens

Solution: Intelligent Token Management

Strategy 1: Document Cap

```
python

PARENT_RETRIEVAL_CAP = 10  # Maximum full documents in Tier 2

# Rank documents by relevance, only expand top N
ranked_docs = sort_by_relevance(tier1_documents)
expand_docs = ranked_docs[:PARENT_RETRIEVAL_CAP]
```

Why 10?

- Average document: 5,000-10,000 tokens
- 10 docs = ~50,000-80,000 tokens
- Stays within quality zone
- Provides sufficient context for complex questions

Strategy 2: Smart Chunking During Expansion

python

```python
def expand_with_budget(doc_id, max_tokens=8000):
    """Expand document but enforce token limit"""
    chunks = fetch_all_chunks(doc_id)

    # Start with chunks that were initially retrieved (highest relevance)
    relevant_chunks = get_initially_retrieved_chunks(doc_id)

    # Add surrounding chunks for context
    expanded = []
    tokens = 0
```

```
    for chunk in chunks:
        chunk_tokens = count_tokens(chunk.text)

        if tokens + chunk_tokens > max_tokens:
            # Budget exceeded, truncate
            break

        expanded.append(chunk)
        tokens += chunk_tokens

    return expanded, tokens
```

Result: Each document gets ~8,000 tokens max, preventing any single document from dominating context.

Strategy 3: Selective Expansion

```
python

# Only expand documents that were cited in Tier 1 draft
cited_docs = extract_citations(tier1_answer)

# Or: Only expand if multiple chunks came from same document
doc_frequency = count_chunks_per_document(tier1_chunks)
expand_docs = [doc for doc, count in doc_frequency.items() if count >= 2]
```

Rationale: If Tier 1 found multiple relevant chunks in a document, that document likely contains comprehensive information worth expanding.

Strategy 4: Progressive Context Building

```
python

def progressive_expansion(query, confidence_threshold=0.85):
    # Tier 1: Start small
    tier1_answer, confidence = query_with_chunks(query, top_k=10)

    if confidence >= confidence_threshold:
        return tier1_answer  # ~8K tokens used

    # Tier 2a: Moderate expansion (most common)
    if confidence >= 0.7:
        expanded_answer = expand_to_full_docs(query, top_n=5)
        return expanded_answer  # ~40K tokens used

    # Tier 2b: Aggressive expansion (rare)
```

```
    if confidence < 0.7:
        full_expansion = expand_to_full_docs(query, top_n=10)
        return full_expansion  # ~80K tokens used
```

Adaptive scaling: Only use expensive full context when necessary.

Strategy 5: Context Compression

For extremely long documents, apply compression:

```python
def compress_document(doc_chunks, query):
    """Keep only query-relevant sections"""

    # Score each chunk for relevance
    scores = []
    for chunk in doc_chunks:
        score = compute_relevance(query, chunk.text)
        scores.append((chunk, score))

    # Keep top 70% by relevance score
    sorted_chunks = sorted(scores, key=lambda x: x[1], reverse=True)
    threshold_idx = int(len(sorted_chunks) * 0.7)

    relevant_chunks = [c for c, s in sorted_chunks[:threshold_idx]]

    # Maintain document structure (sort by original index)
    relevant_chunks.sort(key=lambda c: c.chunk_index)

    return relevant_chunks
```

Result: 30% token reduction while preserving most important content.

Real-World Token Management Example

Query: "Analyze wage patterns across all divisions in 1920-1925"

Tier 1 Retrieval:

```
10 chunks retrieved:
- 3 from Document A (Manusco, Pennsylvania)
- 2 from Document B (Lemley, Ohio)
- 2 from Document C (Barkey, Pennsylvania)
```

- 1 from Document D (Smith, New York)
- 1 from Document E (Jones, Pennsylvania)
- 1 from Document F (Brown, Ohio)

Total tokens: 6,850
Confidence: 0.72 (LOW - insufficient geographic coverage)

Tier 2 Expansion Decision:

```python
# Identify documents with multiple hits (deeper relevance)
expansion_candidates = {
    "Document A": 3 chunks,  # Expand (high relevance)
    "Document B": 2 chunks,  # Expand
    "Document C": 2 chunks,  # Expand
    "Document D": 1 chunk,   # Skip (low relevance)
    "Document E": 1 chunk,   # Skip
    "Document F": 1 chunk    # Skip
}

# Expand top 3 documents only
expand_full_context(["Document A", "Document B", "Document C"])

# Result:
# Document A: 47 chunks → 8,432 tokens (budget enforced)
# Document B: 31 chunks → 7,986 tokens
# Document C: 53 chunks → 8,000 tokens (truncated at budget)
```

Total Tier 2 tokens: 24,418
Grand total: 6,850 (T1) + 24,418 (T2) = 31,268 tokens ✅
Well under 80K limit

Monitoring Token Usage

Debug Output:

```
🔍 [21:24:05] Dispatching to LLM: qwen2.5:32b | 🛢️ 3386 tokens
  ├─ Tier 1 Context: 3,386 tokens
  ├─ System Prompt: 450 tokens
  └─ Total Input: 3,836 tokens

⚖️ [21:24:10] Confidence: 0.73 (Threshold: 0.85)
  └─ Triggering Tier 2 expansion...
```

📖 [21:24:12] Expanding 8 documents to full context
├─ Document A: 8,432 tokens
├─ Document B: 7,986 tokens
├─ Document C: 8,000 tokens (budget capped)
└─ Total expansion: 24,418 tokens

🔍 [21:24:12] Dispatching to LLM: qwen2.5:32b | 🛢️ 28,254 tokens
├─ Tier 2 Context: 24,418 tokens
├─ Tier 1 Summary: 1,200 tokens
├─ System Prompt: 450 tokens
├─ Query: 186 tokens
└─ Total Input: 28,254 tokens ✅ SAFE

⏱️ Query completed in 38.2s (Tier 1: 15.3s, Tier 2: 22.9s)

Benefits of Adversarial + Token Management

1. **Hallucination Prevention**: Second LLM catches factual errors before they reach users
2. **Transparent Confidence**: Users see when answers are uncertain (0.73 vs 0.95)
3. **Adaptive Quality**: Simple questions fast (Tier 1), complex questions thorough (Tier 2)
4. **Cost Control**: Token limits prevent runaway context expansion
5. **Reproducible**: Confidence scores and token counts enable A/B testing
6. **Research-Grade**: Explicit verification meets academic standards

The adversarial architecture transforms a "black box" LLM into a transparent, verifiable research tool that acknowledges uncertainty and actively fights hallucination - essential for scholarly work.

Performance Benchmarks

Hardware Requirements

Minimum (Development):

- CPU: 4 cores
- RAM: 16GB
- Storage: 50GB SSD
- Container Limits: 8GB RAM, 2 CPU cores per service

- Container Limits: 8GB RAM, 2 CPU cores per service

Recommended (Current Setup):

- CPU: Apple M4 (or equivalent 8+ cores)
- RAM: 128GB
- Storage: 500GB NVMe SSD
- GPU: Apple Neural Engine (for embeddings) or L40S (HPC)
- Container Limits: 32GB RAM, 8 CPU cores for Flask service

Storage Breakdown:

- Archives: ~20GB
- MongoDB: ~5GB
- ChromaDB vectors: ~500MB
- Docker images: ~5GB
- Working space: ~10GB

Container Resource Allocation (see Docker section for details):

- MongoDB: 4GB RAM, 2 CPUs
- Flask App: 16-32GB RAM, 4-8 CPUs
- Ollama: Runs on host (GPU access)

API Endpoints

Document Management

```python
# Retrieve document by ID
GET /api/documents/<id>
Response: {document JSON}

# Search documents
POST /api/search
Body: {
```

```
Body: {
  "query": "wage disputes 1920s",
  "use_rag": true,
  "top_k": 10
}
Response: {
  "results": [...],
  "sources": {...}
}
```

RAG Query Interface

python

```
# Simple RAG query
POST /api/rag/query
Body: {
  "question": "What were typical railroad wages in 1925?",
  "mode": "simple"  # or "tiered"
}
Response: {
  "answer": "...",
  "sources": [...],
  "metrics": {
    "retrieval_time": 0.42,
    "llm_time": 15.3,
    "total_time": 15.72,
    "chunks_retrieved": 10,
    "tokens": 3386
  }
}

# Tiered investigation (advanced)
POST /api/rag/investigate
Body: {
  "question": "Analyze wage patterns across all divisions in 1920s",
  "confidence_threshold": 0.85
}
Response: {
  "answer": "...",
  "sources": {...},
  "tiers_used": 2,
  "confidence": 0.73,
  "metrics": [...]
}
```

Person Synthesis

```python
# Generate biographical synthesis
POST /api/persons/synthesize
Body: {
  "person_folder": "RDApp-204897Manusco",
  "model": "qwen2.5:32b"
}
Response: {
  "synthesis": "# Antonio Mancuso...",
  "document_count": 173,
  "processing_time": 245.6
}
```

Document Citation & Linkage System

Purpose

Every piece of information generated by the system is traceable back to specific source documents. This ensures:

- **Academic rigor**: All claims can be verified
- **Research transparency**: Users can examine original sources
- **Data provenance**: Track how conclusions were derived
- **Collaborative research**: Share specific documents with colleagues

Citation Architecture

Document Identification Schema

Every document and chunk has a unique, hierarchical identifier:

```
Document ID: 6939d8063334b77a9b7f0cd4
  ├─ Chunk 0: 6939d8063334b77a9b7f0cd4_chunk_000
  ├─ Chunk 1: 6939d8063334b77a9b7f0cd4_chunk_001
  ├─ Chunk 2: 6939d8063334b77a9b7f0cd4_chunk_002
  └─ ...

Filename: RDApp-204897Manusco173.jpg.json
Collection: Relief Record Scans / Manusco, Antonio
```

Document Metadata Stored:

json


```
{
  "_id": "6939d8063334b77a9b7f0cd4",
  "filename": "RDApp-204897Manusco173.jpg.json",
  "person_folder": "RDApp-204897Manusco",
  "collection_type": "Relief Record Scans",
  "source_image": "RDApp-204897Manusco173.jpg",
  "ocr_confidence": 0.87,
  "document_date": "1923-03-15",
  "document_type": "Certificate of Disablement"
}
```


Citation Flow Through RAG Pipeline

1. Retrieval Stage:

python

```python
# Hybrid retriever returns documents with metadata
chunks = hybrid_retriever.get_relevant_documents(query)

for chunk in chunks:
    print(f"Source: {chunk.metadata['filename']}")
    print(f"Doc ID: {chunk.metadata['document_id']}")
    print(f"Chunk: {chunk.metadata['chunk_index']}")
```

2. Context Assembly:

python

```
# Build context with source attribution
context_parts = []
source_mapping = {}

for chunk in chunks:
    doc_id = chunk.metadata['document_id']
    filename = chunk.metadata['filename']

    context_parts.append(f"--- SOURCE: {filename} (ID: {doc_id}) ---\n{chunk.page_content}")
    source_mapping[filename] = doc_id

# LLM sees:
"""
--- SOURCE: RDApp-204897Manusco173.jpg.json (ID: 6939d8063334b77a9b7f0cd4) ---
[document text]

--- SOURCE: RDApp-225166Lemley127.jpg.json (ID: 6939d8083334b77a9b7f1461) ---
[document text]
"""
```

3. LLM Generation with Source Awareness:

The LLM is explicitly prompted to reference sources:

```
TASK: {user_question}
```

CONTEXT:
--- SOURCE: RDApp-204897Manusco173.jpg.json (ID: 6939d8063334b77a9b7f0cd4) ---
[content]
--- SOURCE: RDApp-225166Lemley127.jpg.json (ID: 6939d8083334b77a9b7f1461) ---
[content]

Provide a comprehensive answer, citing specific documents where relevant.

4. Response with Citations:

The system returns both the answer and the source mapping:

json


```
{
  "answer": "Railroad brakemen in the 1920s earned approximately $1.50-$2.00 per hour...",
  "sources": {
    "RDApp-204897Manusco173.jpg.json": "6939d8063334b77a9b7f0cd4",
    "RDApp-225166Lemley127.jpg.json": "6939d8083334b77a9b7f1461",
    "RDApp-634877Barkey014.jpg.json": "6939d8043334b77a9b7f0654"
  },
  "metrics": {
    "chunks_retrieved": 10,
    "documents_cited": 3,
    "confidence": 0.92
  }
}
```


Frontend Integration

Document Link Generation

URL Schema:

https://app.domain.com/document/{document_id}
https://app.domain.com/document/6939d8063334b77a9b7f0cd4

Link Components in UI:

1. **Inline Citations** (in answer text):

html

Railroad brakemen earned <cite data-doc="6939d8063334b77a9b7f0cd4">$1.50-$2.00 per hour</cite>

- Hover: Shows document preview
- Click: Opens document viewer

2. **Source List** (below answer):

📚 SOURCES CITED:
• RDApp-204897Manusco173.jpg.json [View] [Download] [Share]
• RDApp-225166Lemley127.jpg.json [View] [Download] [Share]
• RDApp-634877Barkey014.jpg.json [View] [Download] [Share]

3. **Document Viewer Modal:**

```
┌─────────────────────────────────────────────────────────┐
│ RDApp-204897Manusco173.jpg                              │
│ Certificate of Disablement - March 15, 1923             │
├─────────────────────────────────────────────────────────┤
│                                                         │
│ [Document Image or OCR Text]                            │
│                                                         │
│ Relevant Section Highlighted:                           │
│ "Wages were not allowed from March 12                   │
│  to April 3, 1923. Weekly wage: $60.00"                 │
│                                                         │
├─────────────────────────────────────────────────────────┤
│ [← Previous] [Download] [Share] [Next →]                │
└─────────────────────────────────────────────────────────┘
```

Advanced Citation Features

1. Chunk-Level Precision:

python

```
# Return not just document, but specific chunk location
{
  "document_id": "6939d8063334b77a9b7f0cd4",
  "chunk_index": 5,
  "chunk_start_char": 4872,
```

```
  "chunk_end_char": 5872,
  "highlighted_text": "Wages were not allowed..."
}
```


Frontend can:

- Scroll directly to relevant section
- Highlight cited passages
- Show context (chunks before/after)

2. Multi-Document Comparison:

```
Compare Documents:
[Doc A: Manusco wages] ↔ [Doc B: Lemley wages] ↔ [Doc C: Barkey wages]

Side-by-side view showing:
- Same form fields across documents
- Wage differences highlighted
- Date ranges compared
```

3. Citation Export:

Users can export citations in multiple formats:

APA:

```
Manusco, A. (1923). Certificate of Disablement (Form 5 No. 98473).
Baltimore & Ohio Railroad Relief Department. Document ID: 6939d8063334b77a9b7f0cd4.
Retrieved from Historical Document Reader.
```

Chicago:

```
Baltimore & Ohio Railroad Relief Department. Certificate of Disablement.
Form 5 No. 98473, Antonio Manusco. March 15, 1923.
Historical Document Reader, document ID 6939d8063334b77a9b7f0cd4.
```

BibTeX:

bibtex

```
@misc{bo_railroad_1923_manusco,
```

```
  author = {{Baltimore \& Ohio Railroad Relief Department}},
  title = {Certificate of Disablement: Antonio Manusco},
  year = {1923},
  month = {March},
  note = {Document ID: 6939d8063334b77a9b7f0cd4},
  url = {https://app.domain.com/document/6939d8063334b77a9b7f0cd4}
}
```

Citation Persistence & Versioning

Problem: Documents might be re-processed, causing chunk IDs to change.

Solution: Stable document IDs with version tracking

python

```
{
  "_id": "6939d8063334b77a9b7f0cd4",  # Stable forever
  "version": 2,
  "version_history": [
    {
      "version": 1,
      "processed_date": "2024-10-15",
      "chunk_count": 47,
      "ocr_model": "llama3.2-vision"
    },
    {
      "version": 2,
      "processed_date": "2024-11-03",
      "chunk_count": 52,
      "ocr_model": "qwen2-vl",
      "changes": "Improved OCR accuracy, additional metadata"
    }
  ]
}
```

Citation Permanence:

- Document URLs never break (`/document/{id}` always works)
- Version parameter for exact reproduction: `/document/{id}?version=1`
- System shows warning if citing older version

Research Workflow Integration

Research Workflow Integration

Example: Building a Research Paper

1. **Initial Query:**

```
User: "What were typical wage ranges for railroad brakemen in Pennsylvania, 1920-1925?"
```

2. **System Response:**

```
Answer: [detailed response]

Sources cited: 15 documents
[Export Citations] [Create Collection] [Add to Research Board]
```

3. **Collection Management:**

```
My Collections > "Pennsylvania Brakemen Wages 1920-1925"
├─ RDApp-204897Manusco173.jpg.json
├─ RDApp-225166Lemley127.jpg.json
├─ RDApp-634877Barkey014.jpg.json
└─ ... (12 more)

[Export All] [Share Collection] [Add Notes] [Generate Bibliography]
```

4. **Collaborative Sharing:**

```
Share Link: https://app.domain.com/collection/wage-study-pa-1920s

Permissions:
☑ View documents
☑ View annotations

☐ Edit annotations
☐ Add documents
```

5. **Citation in Paper:**

```
latex
```

```
\section{Wage Analysis}

Brakemen in Pennsylvania divisions earned between \$1.50 and \$2.00 per hour
during the period 1920-1925, according to relief department records
\cite{bo_railroad_1923_manusco, bo_railroad_1924_lemley}.

\bibliography{my_research}
```

Technical Implementation

Flask Route for Document Viewing:

python

```python
@app.route('/document/<document_id>')
def view_document(document_id):
    # Fetch document from MongoDB
    doc = db.documents.find_one({"_id": ObjectId(document_id)})

    # Get all chunks for this document
    chunks = list(db.document_chunks.find(
        {"document_id": document_id}
    ).sort("chunk_index", 1))

    # Optional: Highlight specific chunk if provided
    highlight_chunk = request.args.get('chunk_index')

    return render_template('document_viewer.html',
        document=doc,
        chunks=chunks,
        highlight=highlight_chunk
    )
```

Citation API Endpoint:

python

```python
@app.route('/api/citations/export', methods=['POST'])
def export_citations():
    document_ids = request.json.get('document_ids')
    format = request.json.get('format', 'apa')  # apa, chicago, bibtex

    citations = generate_citations(document_ids, format)
```

```
    return jsonify({
        'format': format,
        'citations': citations,
        'count': len(citations)
    })
```

Benefits for Historical Research

1. **Verifiability**: Every claim can be traced to primary sources
2. **Transparency**: Research process is documented and reproducible
3. **Collaboration**: Researchers can share exact document sets
4. **Teaching**: Students can see how historians work with primary sources
5. **Publication**: Direct citations to archival materials with stable URLs
6. **Data Provenance**: Track how AI-generated insights derive from sources

This citation system transforms the RAG pipeline from a "black box" into a transparent research tool, maintaining the rigor expected in historical scholarship while leveraging AI efficiency.

Query Performance

Query Type	Retrieval	LLM	Total	Chunks	Tokens
Simple factual	0.2s	8s	8.2s	10	2.5K
Complex analysis	0.4s	15s	15.4s	10	4K
Tiered (T1 only)	0.4s	20s	20.4s	10	4K
Tiered (T1+T2)	2.1s	38s	40.1s	10+10	15K

Model: qwen2.5:32b on M4 Mac Pro

Retrieval Quality Metrics

Metric	Before RAG	After RAG	Improvement
Relevant results	40%	85%	+113%
"No answer found"	30%	5%	-83%

Answer accuracy	60%	90%	+50%
User satisfaction	3.2/5	4.6/5	+44%

(Based on testing with 100 historical queries)

Novel Contributions

What Makes This System Unique

1. **Tiered Investigation Architecture** ⭐⭐⭐
   - LLM self-evaluates confidence
   - Dynamically expands context only when needed
   - Balances cost vs. quality
2. **Historical Document Specialization** ⭐⭐⭐
   - OCR error handling
   - Hierarchical person-based organization
   - Railroad industry-specific entity recognition
   - 1920s wage/disability context understanding
3. **Person Synthesis Pipeline** ⭐⭐⭐
   - Multi-document biographical narrative generation
   - Temporal pattern recognition
   - Career trajectory analysis
   - Network relationship mapping (planned)
4. **Hybrid Retrieval with Domain Adaptation** ⭐⭐

   - Combines semantic + exact match
   - Optimized for historical terminology
   - Handles OCR artifacts and spelling variations

Comparison to Standard RAG

Comparison to Standard RAG

Feature	Standard RAG	This System
Context selection	Fixed top-K	Adaptive (tiered)
Document handling	Chunks only	Small-to-big expansion
Domain	Generic	Historical archives
Entity extraction	Basic NER	Industry-specific + linking
Synthesis	Single-doc summary	Multi-doc biographical narratives
Query answering	One-shot	Self-critique + refinement

Maintenance & Operations

Daily Operations

Monitoring:

```bash
# Check system health
docker compose ps
docker compose logs -f flask_app

# MongoDB health
docker compose exec mongodb mongosh --eval "db.serverStatus()"

# ChromaDB stats
python -c "from vector_store import get_vector_store; print(get_vector_store().get_stats())"
```

Backups:

```bash

# MongoDB
docker compose exec mongodb mongodump --out=/data/backup

# ChromaDB
tar -czf chroma_backup_$(date +%Y%m%d).tar.gz /path/to/chroma_db
```

```
tar -czf chroma_backup_$(date +%Y%m%d).tar.gz /path/to/chroma_db

# Archives (already backed up externally)
```

Common Issues & Solutions

Issue: Slow query responses (> 60s)

```
bash

# Check LLM model
echo $LLM_MODEL  # Should be qwen2.5:32b, NOT gpt-oss:20b

# Reduce context
export HISTORIAN_AGENT_TOP_K=10
export USE_RERANKING=0
```

Issue: "No relevant documents found"

```
bash

# Check vector store
python -c "from vector_store import get_vector_store; \
    stats = get_vector_store().get_stats(); \
    print(f'Chunks indexed: {stats}')"

# If 0, re-run migration
python scripts/embed_existing_documents.py
```

Issue: Out of memory

```
bash

# Reduce batch size
export HISTORIAN_AGENT_TOP_K=5
export PARENT_RETRIEVAL_CAP=5

# Or upgrade RAM / use HPC
```

Future Roadmap

Phase 2: Advanced Features

- [ ] Network analysis visualization
- [ ] Temporal relationship mapping
- [ ] Interactive timeline generation
- [ ] Comparative wage analysis tools
- [ ] Export to scholarly citation formats

Phase 3: Scaling & Optimization

- [ ] HPC batch processing for large queries
- [ ] Query result caching
- [ ] Multi-user access control
- [ ] API rate limiting
- [ ] Advanced monitoring dashboard

Phase 4: Research Integration

- [ ] Integration with external archives
- [ ] Cross-archive entity linking
- [ ] Collaborative annotation tools
- [ ] Research notebook integration
- [ ] Published dataset releases

Technical Debt & Known Limitations

Current Limitations

1. **Token Context Limits**
    - Current: ~80K tokens max
    - Some documents exceed this when fully expanded
    - Mitigation: Tiered retrieval with selective expansion
2. **OCR Error Handling**

    - OCR quality varies significantly
    - Some documents barely readable
    - Mitigation: Multiple OCR passes, manual correction flags

3. **Person Disambiguation**
    - Name variants not fully resolved
    - "John Smith" vs "J. Smith" vs "Jno. Smith"
    - Mitigation: Fuzzy matching, but needs manual review
4. **Query Latency**
    - Current: 15-60 seconds depending on complexity
    - Goal: < 10 seconds
    - Mitigation: Caching, model optimization, GPU acceleration

Technical Debt

1. **Code Organization**
    - Some utility functions duplicated across modules
    - Need centralized configuration management
    - Refactor: Consolidate into shared `utils.py`
2. **Testing Coverage**
    - Integration tests incomplete
    - Need automated regression testing
    - Action: Build test suite with pytest
3. **Documentation**
    - API documentation needs OpenAPI spec
    - Code comments need standardization
    - Action: Generate Sphinx docs

Conclusion

The Historical Document Reader represents a sophisticated integration of traditional database management, modern RAG techniques, and domain-specific AI applications for historical research. The system successfully balances:

- **Performance**: Sub-30-second query responses