Page-Level Metadata Tracking & Advanced RAG Research

Date: November 28, 2025
Status: ✅ Implemented and Deployed

📄 Page-Level Metadata Tracking Implementation

What Was Implemented

Added comprehensive page-level metadata tracking throughout the entire RAG pipeline:

1. PDF Processor Updates (`lib/pdf-processor.ts`)

New Data Structures:

```
export interface PageText {
  pageNumber: number
  text: string
}

export interface TextChunk {
  content: string
  pageNumber: number
  chunkIndex: number
}
```

New Functions:
- `extractTextFromPDF()`: Returns `PageText[]` instead of plain string
- Extracts text page-by-page using `unpdf`
- Handles both array and single-page PDFs
- Tracks which page each text block came from

- `chunkPages()`: Replaces `chunkText()` with page tracking
- Processes each page individually
- Preserves page numbers during chunking
- Logs page-by-page processing statistics

2. Document Upload API (`app/api/documents/upload/route.ts`)

Changes:

```
// OLD: Lost page information
const extractedText = await extractTextFromPDF(buffer)
const chunks = await chunkText(extractedText)

// NEW: Preserves page metadata
const pages = await extractTextFromPDF(buffer) // PageText[]
const chunks = await chunkPages(pages) // TextChunk[]

// Store page numbers in database
await prisma.documentChunk.create({
  data: {
    documentId: document.id,
    content: chunk.content,
    embedding: JSON.stringify(embedding),
    chunkIndex: chunk.chunkIndex,
    pageNumber: chunk.pageNumber, // ✅ Now stored!
  },
})
```

3. RAG Search Functions (`lib/embeddings-v2.ts`)

Updated Return Types:
- `semanticSearch()` : Now returns `pageNumber: number | null`
- `keywordSearch()` : Now returns `pageNumber: number | null`
- `hybridSearch()` : Now returns `pageNumber: number | null`

Database Queries: All search functions now retrieve `pageNumber` from chunks

4. Chat API (`app/api/chat/route.ts`)

Enhanced Context with Page Citations:

```
// OLD: Generic source citation
[Source Document: Federal-Decree-Law-No.-47-of-2022-EN.pdf]

// NEW: Precise page citation
[Source: Federal-Decree-Law-No.-47-of-2022-EN.pdf (Page 48)]
```

Updated System Prompt:
- Instructs LLM to cite document name AND page number
- Example: “According to Federal-Decree-Law-No.-47-of-2022-EN.pdf, Page 48…”

🔄 Impact on Existing Documents

Current Documents

- ❌ **Existing chunks**: `pageNumber = null` (not retroactively filled)
- ✅ **New uploads**: Will have proper page tracking

To Get Page Numbers for Existing Documents

- **Solution**: Re-upload the documents
- **Why**: The chunking algorithm needs to run again with the new page-aware processor

---

🔬 Advanced RAG Tools & Options Research

1. Unstructured.io

✅ TypeScript Support: YES

Official SDK: `unstructured-client` (v0.29.1 as of Nov 2024)

Capabilities:
- 🎯 **PDF Processing**: Advanced layout detection, table extraction, header/footer handling
- 📄 **Multi-format**: PDF, DOCX, XLSX, HTML, Markdown, Images (OCR)
- 🚀 **Partition Endpoint**: Cloud API for document processing
- ⚡ **PDF Page Splitting**: Concurrent processing of large PDFs (up to 15 parallel requests)
- 🔄 **Retry Logic**: Exponential backoff with configurable parameters
- 🎨 **Custom Chunking**: `chunkingStrategy` parameter with options like:
- `by_title` (hierarchical)
- `basic` (fixed-size)
- No chunking (preserve document structure)

Installation:

```
npm install unstructured-client
# or
yarn add unstructured-client
```

Pros:
- ✅ Purpose-built for unstructured documents
- ✅ Handles complex PDFs better (tables, multi-column, headers/footers)
- ✅ Cloud API = no local dependencies
- ✅ Free tier: 1000 pages/day for 14 days

Cons:
- ❌ Requires API key (external dependency)
- ❌ Costs beyond free tier
- ❌ Network latency (API calls)
- ❌ SDK in beta (breaking changes possible)

Use Case:
- Complex legal documents with tables and multi-column layouts
- PDFs with challenging formatting
- When you need enterprise-grade document parsing

---

2. LlamaIndex.TS

✅ TypeScript Support: YES

Official Library: `llamaindex` on npm

Capabilities:
- 📚 **Ingestion Pipeline**: Transformations + Node Parsers
- 🔀 **Multiple Chunking Strategies**:
- `SimpleFileNodeParser`: Auto-detects file type
- `HTMLNodeParser`: HTML-aware splitting

- `JSONNodeParser`: JSON structure preservation
- `RecursiveCharacterTextSplitter`: Language-aware splitting (Python, JS, TS, C, etc.)
- `SemanticSplitter`: Embedding-based semantic boundaries
- `TokenTextSplitter`: Token-aware chunking (Tiktoken)
- `SentenceWindowNodeParser`: Maintains sentence context
- 🎯 **Documents and Nodes**: First-class data structures
- 🔗 **LLM Integration**: Built-in support for OpenAI, Anthropic, Cohere, etc.
- 📊 **Vector Stores**: Pinecone, Weaviate, Chroma, etc.
- 🤖 **Agents & Tools**: Orchestration framework

Installation:

```
npm install llamaindex
# or
yarn add llamaindex
```

Pros:
- ✅ Comprehensive RAG framework (not just chunking)
- ✅ TypeScript-first design
- ✅ Active development and community
- ✅ Many pre-built integrations
- ✅ No API costs (runs locally)

Cons:
- ❌ Heavier framework (more dependencies)
- ❌ Learning curve for full framework
- ❌ May be overkill if you only need chunking

Use Case:
- Building a full RAG application from scratch
- Need advanced orchestration (agents, tools)
- Want a batteries-included solution
- Plan to switch vector stores/LLMs frequently

---

3. Semantic Chunking with Embeddings

✅ TypeScript Support: YES (Multiple implementations)

Concept: Use embeddings to detect semantic boundaries instead of fixed-size chunks

How It Works:
1. Split text into sentences
2. Combine adjacent sentences with context window (buffer)
3. Generate embeddings for each combined sentence
4. Calculate cosine similarity between consecutive embeddings
5. Identify significant drops in similarity (semantic shifts)
6. Create chunk boundaries at these shifts

Libraries:
- **LangChain**: `RecursiveCharacterTextSplitter` with semantic mode

- **Chonkie**: Advanced chunking library (Semantic Chunker, SDPM Chunker, etc.)
- **Custom**: Greg Kamradt's algorithm (TypeScript implementation available)

Example Implementation:

```
import { OpenAIEmbeddings } from '@langchain/openai'
import { SentenceTokenizerNew } from 'natural'

// 1. Split into sentences
const sentences = tokenizer.tokenize(text)

// 2. Create contextual windows
const contextualSentences = sentences.map((sentence, i) => ({
  sentence,
  combined: sentences.slice(i - bufferSize, i + bufferSize + 1).join(' ')
}))

// 3. Generate embeddings
const embeddings = await Promise.all(
  contextualSentences.map(s => embedder.embedQuery(s.combined))
)

// 4. Calculate cosine distances
const distances = embeddings.map((emb, i) =>
  i < embeddings.length - 1 ? cosineSimilarity(emb, embeddings[i + 1]) : 0
)

// 5. Find semantic boundaries (90th percentile threshold)
const threshold = quantile(distances, 0.9)
const boundaries = distances.map((d, i) => d > threshold ? i : -1).filter(i => i !== -
1)

// 6. Create chunks
const chunks = boundaries.map((start, i) => {
  const end = boundaries[i + 1] || sentences.length
  return sentences.slice(start, end).join(' ')
})
```

Pros:
- ✅ Context-aware chunking (keeps related ideas together)
- ✅ Adapts to document structure automatically
- ✅ Better for RAG quality (semantic coherence)
- ✅ Works with any embedding model

Cons:
- ❌ Computationally expensive (embeddings for every sentence)
- ❌ Slower than fixed-size chunking
- ❌ Requires tuning (buffer size, threshold)
- ❌ More complex implementation

Use Case:
- Long documents with varying topics
- When semantic coherence > chunk size uniformity
- Quality > speed priority
- Documents with natural topic shifts

🔌 OpenAI SDK with Abacus.AI

✅ Compatibility: YES!

How It Works

Abacus.AI provides an **OpenAI-compatible API** that you can use with the official OpenAI SDK by simply changing the base URL and API key.

Standard OpenAI SDK Usage:

```
import OpenAI from 'openai'

const client = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
})
```

With Abacus.AI (Custom Endpoint):

```
import OpenAI from 'openai'

const client = new OpenAI({
  apiKey: process.env.ABACUSAI_API_KEY, // Your Abacus.AI key
  baseURL: 'https://api.abacus.ai/v1', // Abacus.AI endpoint
})

// Now use it exactly like OpenAI SDK!
const response = await client.chat.completions.create({
  model: 'gpt-4o-mini', // Access through Abacus.AI
  messages: [{ role: 'user', content: 'Hello!' }],
})
```

Available Models via Abacus.AI

OpenAI Models:
- `o3`, `o4-mini`
- `gpt-4.1`, `gpt-5.1`
- `gpt-4o`, `gpt-4o-mini`
- `gpt-4-128k`, `gpt-4-32k`

Other Providers (via same API):
- Anthropic: Claude 3.5, Claude Opus
- Google: Gemini Pro, Gemini Flash
- Meta: Llama 3.1, Llama 3.2
- Grok: Grok-2

Benefits

Cost Savings:
- Open-source LLMs: **60-120x cheaper** than GPT-4
- Fine-tuned models: **40-50x cheaper** than GPT-4
- GPT-3.5-Turbo: **3-4x cheaper**

Features:
- ✅ Streaming responses
- ✅ Function calling

- ✅ Error handling (same as OpenAI SDK)
- ✅ Retry logic
- ✅ TypeScript support
- ✅ Multiple model providers (one API)

Embeddings

Your Current Setup:

```
// lib/embeddings-v2.ts
const response = await fetch('https://api.abacus.ai/v1/embeddings', {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${process.env.ABACUSAI_API_KEY}`,
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    model: 'text-embedding-3-small',
    input: text,
  }),
})
```

With OpenAI SDK (Cleaner):

```
import OpenAI from 'openai'

const client = new OpenAI({
  apiKey: process.env.ABACUSAI_API_KEY,
  baseURL: 'https://api.abacus.ai/v1',
})

const response = await client.embeddings.create({
  model: 'text-embedding-3-small',
  input: text,
})

const embedding = response.data[0].embedding
```

Advantages:
- ✅ Cleaner code (no manual fetch)
- ✅ Built-in error handling
- ✅ Automatic retries
- ✅ TypeScript types
- ✅ Easier to switch models

🎯 Recommendations

Immediate Actions

1. Keep Current Setup ✅ RECOMMENDED

Your current implementation is **production-ready**:
- ✅ Real embeddings (Abacus.AI API)
- ✅ Hybrid search (semantic + keyword)

- ✅ Page tracking (just implemented!)
- ✅ Query expansion
- ✅ LangChain text splitter

Why it’s good enough:
- Fast and efficient for your scale
- No unnecessary complexity
- Cost-effective (included in Abacus.AI)
- Working well for legal documents

2. Consider Upgrades Only If…

Use Unstructured.io if:
- ❌ Your PDFs have complex tables/layouts
- ❌ Current chunking misses important structure
- ❌ Need better header/footer handling
- ❌ Processing multi-column documents

Use LlamaIndex.TS if:
- ❌ Building a larger RAG application
- ❌ Need agents and orchestration
- ❌ Want to experiment with different approaches
- ❌ Planning to integrate multiple data sources

Use Semantic Chunking if:
- ❌ Current chunks break semantic coherence
- ❌ Quality > speed is critical
- ❌ Documents have natural topic shifts
- ❌ Willing to spend more on compute

Refactor to OpenAI SDK if:
- ❌ Want cleaner embedding generation code
- ❌ Need built-in retry/error handling
- ❌ Planning to switch between models frequently
- ❌ Want better TypeScript types

Priority Assessment

Upgrade	Effort	Benefit	Priority
Page Tracking	✅ Done	High	✅ **DONE**
OpenAI SDK	Low	Medium	🟡 Nice-to-have
Semantic Chunking	Medium	Medium	🟡 If quality issues
Unstructured.io	Medium	High	🟢 If PDFs are complex
LlamaIndex.TS	High	High	🔴 Only if rebuilding

📊 Testing the Page Tracking

How to Test

1. **Upload a new document** (e.g., re-upload Federal Decree Law PDF)
2. **Check server logs** during upload:
   ```

   ✓ Text extracted successfully
     ◦ Total pages: 60
     ◦ Pages with text: 60

Processing page 1 (2345 chars)…
→ Generated 3 chunks from page 1
Processing page 2 (1987 chars)…
→ Generated 2 chunks from page 2
…

Generated embedding for chunk 0 (page 1), length: 1536
Generated embedding for chunk 1 (page 1), length: 1536
```

1. **Ask a question** about a specific topic (e.g., “what does Article 50 specify?”)
2. **Check the response** - should include page citations:
   ```
   According to Federal-Decree-Law-No.-47-of-2022-EN.pdf, Page 48,
     Article 50 specifies the General anti-abuse rule...
   ```
3. **Verify in server logs**:
   ```
   [Source: Federal-Decree-Law-No.-47-of-2022-EN.pdf (Page 48)]
     Article 50: General anti-abuse rule
     ...
   ```

Expected Behavior

OLD (existing documents):

```
[Source: Federal-Decree-Law-No.-47-of-2022-EN.pdf]
```

NEW (newly uploaded documents):

```
[Source: Federal-Decree-Law-No.-47-of-2022-EN.pdf (Page 48)]
```

🔄 Next Steps

Immediate

1. ✅ Test page tracking with a new document upload
2. ✅ Verify page citations appear in chat responses
3. ✅ Re-upload existing documents if page citations are needed

Optional Enhancements

Low Effort, High Value:

- Refactor to use OpenAI SDK for cleaner code
- Add page number filters in chat (“search only pages 40-50”)
- Show page numbers in document preview

Medium Effort, Context-Dependent:

- Implement semantic chunking for better coherence
- Integrate Unstructured.io for complex PDFs
- Add chunk visualization with page numbers

High Effort, Future Expansion:

- Migrate to LlamaIndex.TS for full RAG framework
- Enable pgvector for database-level vector search
- Build custom chunking strategies per document type

📚 Resources

Documentation

- Unstructured.io TypeScript SDK (https://github.com/Unstructured-IO/unstructured-js-client)
- LlamaIndex.TS Documentation (https://ts.llamaindex.ai/)
- OpenAI SDK - Custom Endpoints (https://github.com/openai/openai-node#custom-base-url)
- Abacus.AI API Docs (https://abacus.ai/help/api)
- LangChain Text Splitters (https://js.langchain.com/docs/modules/data_connection/document_transformers/)

Semantic Chunking Resources

- Greg Kamradt’s Semantic Chunking Algorithm (https://github.com/tsensei/Semantic-Chunking-Typescript)
- Chunking Best Practices (https://blog.lancedb.com/chunking-techniques-with-langchain-and-llamaindex/)
- Optimal Chunk Size for Embeddings (https://dev.to/simplr_sh/the-best-way-to-chunk-text-data-for-generating-embeddings-with-openai-models-56c9)

✅ Summary

What Was Done:
- ✅ Implemented page-level metadata tracking throughout RAG pipeline
- ✅ Updated PDF processor to extract and preserve page numbers
- ✅ Modified document upload to store page numbers in database
- ✅ Enhanced RAG search functions to include page metadata
- ✅ Updated chat API to show page citations in responses
- ✅ Researched advanced chunking tools and OpenAI SDK compatibility

Key Insights:
- Your current setup is production-ready and efficient
- Abacus.AI is OpenAI-compatible (can use OpenAI SDK)
- Advanced tools (Unstructured, LlamaIndex) available if needed
- Page tracking is critical for legal citations (now implemented!)

Recommendation: **Stick with current implementation**. It’s working great! Only consider upgrades if you encounter specific limitations.