Retrieval-Augmented Generation (RAG) Integration

Overview

This document outlines the strategy for implementing Retrieval-Augmented Generation (RAG) in the Platform Dashboard. RAG enhances the AI chatbot capabilities by retrieving relevant information from a knowledge base before generating responses, improving accuracy and relevance while reducing hallucinations.

Table of Contents

Introduction

Retrieval-Augmented Generation (RAG) combines retrieval-based and generation-based approaches for AI responses:

1. **Retrieval**: When a user submits a query, the system retrieves relevant information from a knowledge base
2. **Augmentation**: This retrieved information is combined with the user's query
3. **Generation**: The augmented context is sent to the LLM for response generation

Benefits:

- Enhanced factual accuracy using up-to-date information
- Reduced hallucinations by grounding responses in sourced content
- Ability to cite sources of information
- Improved handling of domain-specific queries

Architecture

The RAG implementation will be built on our existing Ollama integration, with the following components:

```
Platform Dashboard RAG Architecture
├── Frontend Components
│   ├── Document Management UI
│   ├── Enhanced Chat Interface
```

```
│   └── Knowledge Base Explorer
├── Backend Services
│   ├── Document Processing Service
│   ├── Vector Database Integration (ChromaDB)
│   ├── Embedding Generation Service
│   └── RAG Query Pipeline
└── Data Storage
    ├── Document Store (File System)
    ├── Vector Database (ChromaDB)
    ├── Document Metadata (PostgreSQL)
    └── RAG Configuration Settings (PostgreSQL)
```

Database Schema Updates

New Tables

1. `vector_stores`
   - Track configured vector database instances

```sql
CREATE TABLE public.vector_stores (
    id UUID DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    store_type VARCHAR(50) NOT NULL,
    connection_string TEXT,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    config JSONB
);
```

2. `document_collections`
   - Organize documents into collections (e.g., "Technical Documentation", "Policies")

```sql
CREATE TABLE public.document_collections (
    id UUID DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    user_id UUID NOT NULL REFERENCES public.users(id) ON DELETE CASCADE,
    vector_store_id UUID REFERENCES public.vector_stores(id) ON DELETE SET
NULL,
    embedding_model VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_active BOOLEAN DEFAULT TRUE,
    metadata JSONB
);
```

3. `documents`
    - Enhanced version of existing `pdfs` table to support all document types

```
CREATE TABLE public.documents (
    id UUID DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    user_id UUID NOT NULL REFERENCES public.users(id) ON DELETE CASCADE,
    collection_id UUID REFERENCES public.document_collections(id) ON DELETE
SET NULL,
    title VARCHAR(255) NOT NULL,
    file_path TEXT NOT NULL,
    file_name VARCHAR(255) NOT NULL,
    file_type VARCHAR(50) NOT NULL,
    content_text TEXT,
    content_hash VARCHAR(64),
    vector_id VARCHAR(255),
    processing_status VARCHAR(20) DEFAULT 'pending' NOT NULL, -- 'pending',
'processing', 'processed', 'failed'
    is_indexed BOOLEAN DEFAULT FALSE,
    chunk_count INTEGER DEFAULT 0,
    uploaded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    processed_at TIMESTAMP,
    indexed_at TIMESTAMP,
    last_accessed_at TIMESTAMP,
    metadata JSONB
);
```

4. `document_chunks`
    - Store individual chunks of documents for fine-grained retrieval

```
CREATE TABLE public.document_chunks (
    id UUID DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    document_id UUID NOT NULL REFERENCES public.documents(id) ON DELETE
CASCADE,
    chunk_index INTEGER NOT NULL,
    content TEXT NOT NULL,
    vector_id VARCHAR(255),
    embedding VECTOR(1536),
    token_count INTEGER,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    metadata JSONB
);
```

5. `rag_settings`
    - Configure RAG behavior and parameters

```sql
CREATE TABLE public.rag_settings (
    id SERIAL PRIMARY KEY,
    embedding_model VARCHAR(255) DEFAULT 'ollama/nomic-embed-text',
    chunk_size INTEGER DEFAULT 1000,
    chunk_overlap INTEGER DEFAULT 200,
    similarity_top_k INTEGER DEFAULT 4,
    search_type VARCHAR(20) DEFAULT 'similarity',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    config JSONB
);
```

Updating Existing Tables

1. `messages` - Add references to retrieved document chunks

```sql
ALTER TABLE public.messages ADD COLUMN retrieved_chunks JSONB;
```

2. `chat_sessions` - Add option to enable/disable RAG for a session

```sql
ALTER TABLE public.chat_sessions ADD COLUMN use_rag BOOLEAN DEFAULT TRUE;
ALTER TABLE public.chat_sessions ADD COLUMN rag_collections JSONB;
```

Implementation Steps

Phase 1: Foundation (Weeks 1-2)

1. **Database Setup**
    - Create new tables in PostgreSQL
    - Create migration scripts
    - Update database documentation
2. **Vector Database Integration**
    - Install ChromaDB locally or via Docker
    - Implement connection service
    - Create CRUD operations for vectors
    - Implement configuration management
3. **Document Processing Service**
    - Create file upload and processing pipeline
    - Implement text extraction from PDFs, DOCXs, etc.
    - Implement text chunking strategies
    - Create embedding generation service using Ollama models

Phase 2: Core RAG Implementation (Weeks 3-4)

1. **Query Pipeline**

    - Implement query embedding generation
    - Develop vector search functionality
    - Create context assembly for LLM prompts
    - Implement citation tracking

2. **Ollama Integration**

    - Enhance Ollama service to support RAG
    - Implement prompt templates for RAG queries
    - Create specialized chat completion with context

3. **API Endpoints**

    - Implement document management endpoints
    - Create collection management endpoints
    - Develop RAG-enabled chat endpoints

Phase 3: Frontend and User Experience (Weeks 5-6)

1. **Document Management UI**

    - Create document upload interface
    - Implement collection management
    - Develop processing status indicators

2. **Enhanced Chat Interface**

    - Update chat UI to show citations
    - Implement RAG toggle option
    - Create source preview functionality

3. **Knowledge Base Explorer**

    - Develop interface to browse uploaded documents
    - Create collection and document search
    - Implement document preview

Vector Database Integration

ChromaDB will be used as the vector database for storing and retrieving document embeddings:

1. **Installation**

    ```
    npm install chromadb
    ```

2. **Configuration**

- Local ChromaDB instance (development)
- Persistent store using PersistentClient
- Collection per document group

3. **Integration Points**
   - Document upload pipeline (embedding generation)
   - Query processing (similarity search)
   - Collection management (CRUD operations)

Core Functionality

```
// src/services/vectorStoreService.js
const { ChromaClient } = require('chromadb');

class VectorStoreService {
  constructor(config) {
    this.client = new ChromaClient(config.host, config.port);
    this.defaultEmbeddingFunction =
this.getEmbeddingFunction(config.embeddingModel);
  }

  async getCollection(name, embeddingFunction = this.defaultEmbeddingFunction) {
    try {
      return await this.client.getCollection(name, embeddingFunction);
    } catch (error) {
      return await this.client.createCollection(name, { embeddingFunction });
    }
  }

  getEmbeddingFunction(modelName) {
    // Integration with Ollama embedding models
    return {
      generate: async (texts) => {
        // Call Ollama embedding endpoint
        const embeddings = await ollamaService.generateEmbeddings(modelName,
texts);
        return embeddings;
      }
    };
  }

  // Other methods for CRUD operations, search, etc.
}
```

Document Processing Pipeline

The document processing pipeline will handle various document types and prepare them for RAG:

Directory Structure

```
productdemo/
├── Documents/          # Storage for uploaded document files
│   ├── user_id_1/      # Organized by user ID
│   │   ├── doc_id_1/   # Each document gets its own folder
│   │   │   └── original.pdf
│   │   └── doc_id_2/
│   │       └── original.docx
│   └── user_id_2/
│       └── ...
│
├── Embeddings/         # Storage for document embeddings
    ├── user_id_1/      # Organized by user ID
    │   ├── doc_id_1/   # Each document gets its own folder
    │   │   ├── metadata.json    # Document metadata including hash
    │   │   ├── chunks.json      # Text chunks
    │   │   └── embeddings.bin   # Binary embedding vectors
    │   └── doc_id_2/
    │       └── ...
    └── user_id_2/
        └── ...
```

Processing Flow

1. **Upload Phase**
   - File validation and sanitization
   - Document metadata extraction
   - Storage in file system (`Documents/user_id/doc_id/original.[ext]`)
   - Entry created in database with status "pending"
2. **Processing Phase**
   - Text extraction based on document type
   - Content chunking with configurable overlap
   - Embedding generation using Ollama models
   - Vector storage in ChromaDB
   - Status updates in database
   - Storage of embeddings in `Embeddings/user_id/doc_id/` directory
3. **Indexing Phase**
   - Metadata indexing
   - Vector indexing optimization
   - Relationship mapping for collections

Deduplication Strategy

To avoid reprocessing identical documents:

1. **Hash Verification**

- When a document is uploaded, calculate its content hash
- Check if a document with the same hash exists in the user's collection
- If match found, verify metadata similarity (optional)

2. **Reuse Existing Embeddings**

    - If duplicate detected, link to existing embeddings in database
    - Update document record with reference to existing vector IDs
    - Skip processing pipeline, mark as "processed" immediately

3. **Partial Updates**

    - For similar but not identical documents, consider partial updates
    - Only process and embed changed sections (future enhancement)

Text Extraction and Chunking

```
// src/services/documentProcessingService.js
const fs = require('fs');
const path = require('path');
const pdf = require('pdf-parse');
const mammoth = require('mammoth');

class DocumentProcessingService {
  constructor(config, vectorStoreService) {
    this.config = config;
    this.vectorStoreService = vectorStoreService;
    this.chunkSize = config.chunkSize || 1000;
    this.chunkOverlap = config.chunkOverlap || 200;
  }

  async processDocument(document) {
    // Extract text based on document type
    const text = await this.extractText(document);

    // Chunk text into segments
    const chunks = this.chunkText(text);

    // Generate embeddings and store in vector db
    await this.vectorizeChunks(document.id, chunks, document.collection_id);

    return {
      documentId: document.id,
      chunkCount: chunks.length,
      status: 'processed'
    };
  }

  async extractText(document) {
    const filePath = document.file_path;
    const fileType = document.file_type.toLowerCase();
```

```
    switch (fileType) {
      case 'pdf':
        return await this.extractFromPDF(filePath);
      case 'docx':
        return await this.extractFromDOCX(filePath);
      case 'txt':
        return await this.extractFromTXT(filePath);
      // Add other document types as needed
      default:
        throw new Error(`Unsupported file type: ${fileType}`);
    }
  }

  // Methods for specific file type extraction
  async extractFromPDF(filePath) {
    const dataBuffer = fs.readFileSync(filePath);
    const data = await pdf(dataBuffer);
    return data.text;
  }

  async extractFromDOCX(filePath) {
    const result = await mammoth.extractRawText({ path: filePath });
    return result.value;
  }

  async extractFromTXT(filePath) {
    return fs.readFileSync(filePath, 'utf8');
  }

  // Methods for chunking
  chunkText(text) {
    const chunks = [];
    let startIndex = 0;

    while (startIndex < text.length) {
      // Calculate end index with consideration for overlap
      const endIndex = Math.min(startIndex + this.chunkSize, text.length);
      chunks.push(text.slice(startIndex, endIndex));

      // Move start index forward, accounting for overlap
      startIndex += this.chunkSize - this.chunkOverlap;

      // Ensure we make progress even with large overlap
      if (startIndex <= (endIndex - this.chunkSize/2)) {
        startIndex = endIndex - this.chunkOverlap;
      }
    }

    return chunks;
  }

  // Methods for vectorization and deduplication
  async vectorizeChunks(documentId, chunks, collectionId) {
    // Generate embeddings for each chunk
```

```
    const embeddings = await this.generateEmbeddings(chunks);

    // Store chunks and embeddings
    await this.storeChunksAndEmbeddings(documentId, chunks, embeddings);

    // Update document status
    await this.updateDocumentStatus(documentId, chunks.length);

    return chunks.length;
  }

  async checkForDuplicates(document) {
    // Calculate content hash if not already done
    if (!document.content_hash) {
      const text = await this.extractText(document);
      document.content_hash = this.calculateContentHash(text);
      await this.updateDocumentHash(document.id, document.content_hash);
    }

    // Check for existing document with same hash
    const existingDoc = await this.findDocumentByHash(
      document.content_hash,
      document.user_id
    );

    if (existingDoc) {
      console.log(`Duplicate document detected: ${document.id} matches
${existingDoc.id}`);

      // Link to existing embeddings
      await this.linkToExistingEmbeddings(document.id, existingDoc.id);

      // Mark as processed without running the pipeline
      await this.updateDocumentStatus(
        document.id,
        existingDoc.chunk_count,
        'processed',
        'Used existing embeddings from duplicate document'
      );

      return true; // Duplicate found
    }

    return false; // No duplicate found
  }

  calculateContentHash(text) {
    const crypto = require('crypto');
    return crypto.createHash('sha256').update(text).digest('hex');
  }
}
```

Frontend Integration

The frontend will be updated to support RAG features:

1. **Document Management Page**
    - Upload interface with drag-and-drop
    - Collection management
    - Processing status and statistics
2. **Chat Enhancements**
    - Toggle for RAG mode
    - Collection selector for context scope
    - Citation display in messages
    - Source preview on citation click
3. **Knowledge Base Explorer**
    - Document browser with search and filters
    - Collection management interface
    - Document preview and metadata display

Component Updates

```
// client/src/components/chat/ChatInput.tsx
// Add RAG options to ChatInput component

const ChatInput: React.FC<ChatInputProps> = ({
  onSendMessage,
  isLoading,
  useRAG,
  onToggleRAG,
  selectedCollections,
  onSelectCollections
}) => {
  // Existing code...

  return (
    <div className="chat-input-container">
      {/* Existing input elements */}

      <div className="rag-controls">
        <Switch
          isChecked={useRAG}
          onChange={onToggleRAG}
          label="Use Knowledge Base"
        />

        {useRAG && (
          <CollectionSelector
            selectedCollections={selectedCollections}
```

```
            onChange={onSelectCollections}
          />
        )}
      </div>
    </div>
  );
};
```

Backend Services

The backend will require several new services:

1. `ragService.js`: Orchestrates the RAG process
2. `vectorStoreService.js`: Manages vector database operations
3. `documentProcessingService.js`: Handles document processing
4. `embeddingService.js`: Generates embeddings using Ollama

RAG Service

```
// src/services/ragService.js
class RAGService {
  constructor(
    vectorStoreService,
    documentService,
    ollamaService,
    config
  ) {
    this.vectorStoreService = vectorStoreService;
    this.documentService = documentService;
    this.ollamaService = ollamaService;
    this.config = config;
  }

  async queryWithRAG(query, model, collectionIds, options = {}) {
    // Generate embedding for the query
    const queryEmbedding = await this.getQueryEmbedding(query);

    // Retrieve relevant chunks from vector store
    const relevantChunks = await this.retrieveRelevantChunks(
      queryEmbedding,
      collectionIds,
      options.topK || 4
    );

    // Format context from retrieved chunks
    const formattedContext = this.formatContext(relevantChunks);

    // Generate augmented response using Ollama
    const response = await this.ollamaService.chatWithContext(
      model,
      query,
```

```
      formattedContext,
      options.systemPrompt
    );

    // Add citation metadata to response
    return this.addCitations(response, relevantChunks);
  }

  // Helper methods for embedding, retrieval, formatting, etc.
}
```

API Endpoints

New API endpoints will be added to support RAG functionality:

Document Management

```
// src/routes/documents.js
const express = require('express');
const router = express.Router();
const multer = require('multer');
const { isAuthenticated } = require('../middleware/auth');

module.exports = function(documentService, ragService) {
  // Configure multer for file uploads
  const upload = multer({
    dest: 'uploads/',
    limits: { fileSize: 50 * 1024 * 1024 } // 50MB limit
  });

  // Upload document
  router.post('/upload', isAuthenticated, upload.single('document'), async (req,
res) => {
    try {
      // Create document record
      const document = await documentService.createDocument({
        file: req.file,
        userId: req.user.id,
        collectionId: req.body.collectionId,
        title: req.body.title || req.file.originalname,
        metadata: req.body.metadata ? JSON.parse(req.body.metadata) : {}
      });

      // Save file to Documents directory
      const userDir = path.join('Documents', document.user_id);
      const docDir = path.join(userDir, document.id);

      // Create directories if they don't exist
      if (!fs.existsSync(userDir)) fs.mkdirSync(userDir, { recursive: true });
      if (!fs.existsSync(docDir)) fs.mkdirSync(docDir, { recursive: true });
```

```
      // Move uploaded file to final destination
      const originalExt = path.extname(document.file_name);
      const finalPath = path.join(docDir, `original${originalExt}`);
      fs.renameSync(req.file.path, finalPath);

      // Update document with final path
      await documentService.updateDocumentPath(document.id, finalPath);

      // Check for duplicates before processing
      const isDuplicate = await documentService.checkForDuplicates(document.id);

      if (!isDuplicate) {
        // Start processing in background if not a duplicate
        documentService.processDocumentAsync(document.id);
      }

      res.status(201).json({
        ...document,
        isDuplicate
      });
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  });

  // Other document management endpoints
  // GET /documents
  // GET /documents/:id
  // DELETE /documents/:id
  // etc.

  return router;
};
```

RAG-Enabled Chat

```
// src/routes/chatbot.js (updated)
router.post('/chat-rag', isAuthenticated, async (req, res) => {
  try {
    const { message, sessionId, modelId, useRAG, collectionIds } = req.body;

    // Create chat session if needed
    let session;
    if (sessionId) {
      session = await getSessionById(sessionId, req.user.id);
      if (!session) {
        return res.status(404).json({ error: 'Session not found' });
      }
    } else {
      session = await createNewSession(req.user.id, message.substring(0, 50));
    }
```

```
    // Save user message
    const messageRecord = await saveMessage(req.user.id, message, null,
session.id);

    // If RAG is enabled, use RAG service for response
    let response;
    if (useRAG && collectionIds && collectionIds.length > 0) {
      response = await ragService.queryWithRAG(
        message,
        modelId,
        collectionIds,
        { includeCitations: true }
      );

      // Update message with retrieved chunks
      await updateMessageWithChunks(messageRecord.id, response.citations);
    } else {
      // Regular Ollama chat without RAG
      response = await ollamaService.chat(modelId, [{
        role: 'user',
        content: message
      }]);
    }

    // Update message with AI response
    await updateMessageResponse(messageRecord.id, response.text);

    res.json({
      sessionId: session.id,
      messageId: messageRecord.id,
      response: response.text,
      citations: response.citations || [],
      model: modelId
    });
  } catch (error) {
    console.error('Error in chat-rag endpoint:', error);
    res.status(500).json({ error: error.message });
  }
});
```

Testing and Validation

The RAG implementation will be tested using:

1. **Unit Tests**

   - Test document processing pipeline
   - Test vector store operations
   - Test chunking and embedding generation

2. **Integration Tests**

- Test end-to-end document upload to RAG query
- Test various document types and sizes
- Test vector database connection reliability

3. **Performance Testing**

   - Measure retrieval latency for various collection sizes
   - Test embedding generation performance
   - Evaluate response quality compared to non-RAG responses

4. **User Testing**

   - Validate relevance of retrieved information
   - Test usability of document management interface
   - Gather feedback on citation presentation

Embedding Storage and Retrieval

Storage Format

The embeddings will be stored in both the filesystem and ChromaDB:

1. **Filesystem Storage**

   - Location: `Embeddings/user_id/doc_id/`
   - Files:
     - `metadata.json`: Document metadata including hash, title, etc.
     - `chunks.json`: Array of text chunks with positions and metadata
     - `embeddings.bin`: Binary file containing embedding vectors

2. **ChromaDB Storage**

   - Collection per user or document group
   - Each document chunk stored with:
     - ID: `doc_id:chunk_index`
     - Embedding: Vector representation
     - Metadata: Document info, chunk position, etc.

Embedding Generation

```
// src/services/embeddingService.js
class EmbeddingService {
  constructor(ollamaService, config) {
    this.ollamaService = ollamaService;
    this.embeddingModel = config.embeddingModel || 'nomic-embed-text';
  }

  async generateEmbeddings(texts) {
    if (!Array.isArray(texts)) {
      texts = [texts];
    }
```

```
    const embeddings = [];

    // Process in batches to avoid overwhelming the Ollama server
    const batchSize = 10;
    for (let i = 0; i < texts.length; i += batchSize) {
      const batch = texts.slice(i, i + batchSize);
      const batchEmbeddings = await Promise.all(
        batch.map(text =>
this.ollamaService.generateEmbedding(this.embeddingModel, text))
      );
      embeddings.push(...batchEmbeddings);
    }

    return embeddings;
  }

  async storeEmbeddings(documentId, userId, chunks, embeddings) {
    // Create directory structure
    const userDir = path.join('Embeddings', userId);
    const docDir = path.join(userDir, documentId);

    if (!fs.existsSync(userDir)) fs.mkdirSync(userDir, { recursive: true });
    if (!fs.existsSync(docDir)) fs.mkdirSync(docDir, { recursive: true });

    // Store chunks
    const chunksWithMetadata = chunks.map((chunk, index) => ({
      index,
      content: chunk,
      token_count: this.estimateTokenCount(chunk)
    }));

    fs.writeFileSync(
      path.join(docDir, 'chunks.json'),
      JSON.stringify(chunksWithMetadata, null, 2)
    );

    // Store embeddings in binary format
    const embeddingBuffer = this.convertEmbeddingsToBuffer(embeddings);
    fs.writeFileSync(path.join(docDir, 'embeddings.bin'), embeddingBuffer);

    // Store metadata
    const metadata = {
      document_id: documentId,
      user_id: userId,
      chunk_count: chunks.length,
      embedding_model: this.embeddingModel,
      created_at: new Date().toISOString(),
      dimensions: embeddings[0].length
    };

    fs.writeFileSync(
      path.join(docDir, 'metadata.json'),
      JSON.stringify(metadata, null, 2)
```

```
    );

    return {
      path: docDir,
      metadata
    };
  }

  // Helper methods
  convertEmbeddingsToBuffer(embeddings) {
    // Convert array of embedding vectors to binary buffer
    const dimensions = embeddings[0].length;
    const buffer = Buffer.alloc(embeddings.length * dimensions * 4); // 4 bytes
per float32

    embeddings.forEach((embedding, embIndex) => {
      embedding.forEach((value, valueIndex) => {
        buffer.writeFloatLE(value, (embIndex * dimensions + valueIndex) * 4);
      });
    });

    return buffer;
  }

  loadEmbeddingsFromBuffer(buffer, count, dimensions) {
    // Load embeddings from binary buffer
    const embeddings = [];

    for (let i = 0; i < count; i++) {
      const embedding = [];
      for (let j = 0; j < dimensions; j++) {
        embedding.push(buffer.readFloatLE((i * dimensions + j) * 4));
      }
      embeddings.push(embedding);
    }

    return embeddings;
  }

  estimateTokenCount(text) {
    // Simple estimation: ~4 characters per token
    return Math.ceil(text.length / 4);
  }
}
```

Retrieval Process

When a user query is processed:

1. **Query Embedding**

```
// Generate embedding for user query
const queryEmbedding = await embeddingService.generateEmbeddings(query);
```

2. **Vector Search**

```
// Search ChromaDB for similar chunks
const results = await vectorStoreService.search(
  collectionId,
  queryEmbedding,
  {
    limit: topK,
    includeMetadata: true,
    includeEmbeddings: false
  }
);
```

3. **Context Assembly**

```
// Format retrieved chunks for LLM context
const context = results.map(result => {
  return {
    content: result.metadata.content,
    source: result.metadata.source,
    document_title: result.metadata.document_title,
    document_id: result.metadata.document_id,
    similarity: result.similarity
  };
});

// Format context for LLM prompt
const formattedContext = formatContextForLLM(context);
```

Future Enhancements

1. **Advanced Retrieval Methods**
    - Hybrid search (keyword + vector)
    - Re-ranking of retrieved chunks
    - Multi-query retrieval strategies
2. **Document Processing Enhancements**
    - OCR for scanned documents and images
    - Table extraction and structured data handling
    - Automatic metadata extraction
3. **Agent Integration**

- Combine RAG with Ollama's function calling
- Create specialized agents with domain knowledge
- Implement autonomous research workflows

4. **Multi-Model Support**

    - Use different models for embedding vs. generation
    - Support multiple embedding spaces
    - Model-specific prompt templates

5. **Expanded Knowledge Sources**

    - Web crawling and automatic updates
    - API integrations for external knowledge bases
    - Database querying capabilities

6. **Embedding Optimization**

    - Compression techniques for embeddings
    - Quantization to reduce storage requirements
    - Incremental updates for large documents

Implementation Timeline

Phase	Timeframe	Key Deliverables
1: Foundation	Weeks 1-2	Database schema, Vector DB integration, Document processing
2: Core RAG	Weeks 3-4	Query pipeline, Ollama integration, API endpoints
3: Frontend	Weeks 5-6	Document management UI, Enhanced chat, Knowledge explorer
4: Testing & Refinement	Weeks 7-8	Testing, Performance optimization, Documentation