Safe Appeals Navigator - Complete RAG Implementation Guide

Table of Contents

Executive Summary

This comprehensive guide provides detailed instructions for implementing Retrieval-Augmented Generation (RAG) capabilities in Safe Appeals Navigator using Chroma vector database and SQLite for metadata storage. The implementation includes a Policy Manual Dashboard, document indexing services, AI agent tool calls, and complete workspace organization features.

Table of Contents

12. Testing Strategy
13. Deployment & Build Process
14. Troubleshooting Guide

Project Architecture Overview

System Components

The RAG implementation consists of several interconnected components:

- **Index Service**: Handles document ingestion, text extraction, and dual storage (Chroma + SQLite)
- **Agent Service**: Provides AI-powered tool calls for document organization and querying
- **Policy Dashboard**: Dedicated interface for policy manual exploration and search
- **RAG Chains**: LangChain-based retrieval systems for both policy and workspace documents
- **File Organization**: AI-driven automatic categorization and folder structure creation

Data Flow Architecture

```
graph TB
    A[Document Upload] --&gt; B[MIME Type Detection]
    B --&gt; C[Text Extraction Service]
    C --&gt; D[Text Chunking &amp;amp; Embedding]
    D --&gt; E[Chroma Vector Store]
    D --&gt; F[SQLite Metadata DB]

    G[User Query] --&gt; H[RAG Chain]
    H --&gt; E
    H --&gt; I[LLM Processing]
    I --&gt; J[Response Generation]

    K[Workspace Organization] --&gt; L[AI Classification]
    L --&gt; E
    L --&gt; M[File System Operations]
```

Technology Stack

- **Frontend**: VS Code Extension API, TypeScript, React (for webviews)
- **Backend**: Node.js, Electron
- **Vector Database**: Chroma (TypeScript client)
- **Metadata Storage**: SQLite (better-sqlite3)
- **AI Framework**: LangChain SDK, OpenAI API
- **Document Processing**: PDF.js, Mammoth.js
- **Build System**: VS Code Extension Build Tools

Development Environment Setup

Prerequisites Installation

```
# Node.js version management (use Node 20.x)<a></a>
nvm install 20.18.2
nvm use 20.18.2

# Global dependencies<a></a>
npm install -g @vscode/vsce typescript

# Verify installations<a></a>
node --version  # Should show v20.18.2
npm --version   # Should show compatible version
```

Project Dependencies

Create a comprehensive `package.json` with all required dependencies:

```
{
  "name": "safe-appeals-navigator",
  "displayName": "Safe Appeals Navigator",
  "version": "1.0.0",
  "description": "AI-powered document editor for workers compensation appeals",
  "engines": {
    "vscode": "^1.90.0",
    "node": "&gt;=20.0.0"
  },
  "categories": ["Other"],
  "activationEvents": [
    "onStartupFinished"
  ],
  "main": "./out/extension.js",
  "dependencies": {
    "@langchain/core": "^0.2.15",
    "@langchain/openai": "^0.2.0",
    "chromadb": "^1.8.1",
    "better-sqlite3": "^9.0.0",
    "openai": "^4.52.0",
    "pdfjs-dist": "^4.0.269",
    "mammoth": "^1.6.0",
    "node-fetch": "^3.3.2",
    "uuid": "^9.0.1",
    "winston": "^3.10.0",
    "marked": "^9.1.0"
  },
  "devDependencies": {
    "@types/node": "^20.5.0",
    "@types/vscode": "^1.90.0",
    "@types/better-sqlite3": "^7.6.4",
    "@types/uuid": "^9.0.2",
    "typescript": "^5.2.2",
    "ts-node": "^10.9.1",
    "@typescript-eslint/eslint-plugin": "^6.4.0",
    "@typescript-eslint/parser": "^6.4.0",
    "eslint": "^8.47.0"
  },
  "scripts": {
    "compile": "tsc -p ./",
    "watch": "tsc -watch -p ./",
    "package": "vsce package",
    "test": "node ./out/test/runTest.js"
  }
}
```

Directory Structure Setup

Create the complete project structure:

```
safe-appeals-navigator/
├── package.json
├── tsconfig.json
├── .vscodeignore
├── README.md
├── CHANGELOG.md
├── LICENSE
├── src/
│   ├── extension.ts                # Main extension entry point
│   ├── services/
│   │   ├── indexService.ts         # Core indexing functionality
│   │   ├── agentService.ts         # AI agent operations
│   │   ├── ragService.ts           # RAG chain management
│   │   ├── fileService.ts          # File operations &amp;amp; MIME handling
│   │   └── configService.ts        # Configuration management
```

```
│   ├── dashboard/
│   │   ├── policyDashboard.ts      # Policy manual interface
│   │   ├── workspaceDashboard.ts   # Workspace insights
│   │   └── components/
│   │       ├── searchWidget.ts     # Search interface components
│   │       └── documentViewer.ts   # Document display components
│   ├── commands/
│   │   ├── uploadCommands.ts       # File upload handlers
│   │   ├── queryCommands.ts        # Search and query handlers
│   │   └── organizationCommands.ts # File organization commands
│   ├── utils/
│   │   ├── logger.ts               # Logging utilities
│   │   ├── errorHandler.ts         # Error handling
│   │   └── pathResolver.ts         # Path resolution utilities
│   └── types/
│       ├── index.ts                # Type definitions
│       └── database.ts             # Database schema types
├── resources/
│   ├── templates/
│   │   └── folderStructure.json    # Default folder organization
│   ├── icons/
│   │   ├── policy.svg
│   │   ├── workspace.svg
│   │   └── search.svg
│   └── webview/
│       ├── dashboard.html          # Dashboard HTML templates
│       ├── styles.css              # Dashboard styling
│       └── scripts.js              # Dashboard JavaScript
├── test/
│   ├── suite/
│   │   ├── extension.test.ts
│   │   ├── indexService.test.ts
│   │   └── ragService.test.ts
│   └── runTest.ts
└── out/                           # Compiled JavaScript output
```

TypeScript Configuration

Create `tsconfig.json` with comprehensive settings:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "ES2022",
    "lib": ["ES2022"],
    "outDir": "out",
    "rootDir": "src",
    "sourceMap": true,
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "moduleResolution": "node",
    "declaration": true,
    "declarationMap": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "skipLibCheck": true,
    "resolveJsonModule": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "out", "test/**/*"]
}
```

Database Configuration

User Data Directory Setup

src/utils/pathResolver.ts

```
import * as vscode from 'vscode';
import * as path from 'path';
import * as fs from 'fs/promises';
import { app } from 'electron';

export class PathResolver {
  private static _instance: PathResolver;
  private _userDataPath: string;
  private _storageDir: string;
  private _chromaDir: string;
  private _sqlitePath: string;

  private constructor() {
    // Use VS Code's global storage path if available, otherwise fall back to Electron
    this._userDataPath = vscode.env.globalStorageUri?.fsPath || app?.getPath('userData') || process.env.APP
    this._storageDir = path.join(this._userDataPath, 'safe-appeals-navigator', 'storage');
    this._chromaDir = path.join(this._storageDir, 'chroma');
    this._sqlitePath = path.join(this._storageDir, 'workspace.db');
  }

  public static getInstance(): PathResolver {
    if (!PathResolver._instance) {
      PathResolver._instance = new PathResolver();
    }
    return PathResolver._instance;
  }

  public get userDataPath(): string {
    return this._userDataPath;
  }

  public get storageDir(): string {
    return this._storageDir;
  }

  public get chromaDir(): string {
    return this._chromaDir;
  }

  public get sqlitePath(): string {
    return this._sqlitePath;
  }

  public async ensureDirectoriesExist(): Promise&lt;void&gt; {
    try {
      await fs.mkdir(this._storageDir, { recursive: true });
      await fs.mkdir(this._chromaDir, { recursive: true });
    } catch (error) {
      throw new Error(`Failed to create storage directories: ${error}`);
    }
  }

  public async validatePaths(): Promise&lt;boolean&gt; {
    try {
      await fs.access(this._storageDir);
      await fs.access(this._chromaDir);
      return true;
    } catch {
      return false;
    }
  }
}
```

Database Schema Definition

src/types/database.ts

```
export interface DocumentRecord {
  id: string;
  filename: string;
  filepath: string;
  filetype: string;
  filesize: number;
  uploadedAt: string;
  lastIndexed: string;
  checksum?: string;
  metadata?: string; // JSON string of additional metadata
}

export interface ChunkRecord {
  chunkId: string;
  docId: string;
  text: string;
  chunkIndex: number;
  embedding?: Float32Array;
  tokens?: number;
}

export interface PolicySection {
  sectionId: string;
  title: string;
  level: number;
  parentId?: string;
  docId: string;
  pageNumber?: number;
  chunkIds: string[];
}

export interface WorkspaceConfig {
  id: string;
  name: string;
  rootPath: string;
  folderStructure: string; // JSON string of folder organization rules
  lastOrganized: string;
  totalDocuments: number;
  indexedDocuments: number;
}

export interface SearchHistory {
  id: string;
  query: string;
  collection: 'policy_manual' | 'workspace_docs';
  timestamp: string;
  resultCount: number;
  responseTime: number;
}
```

Index Service Implementation

Core Index Service

src/services/indexService.ts

```
import { ChromaApi, OpenAIEmbeddingFunction, Collection } from 'chromadb';
import Database from 'better-sqlite3';
import * as vscode from 'vscode';
import * as path from 'path';
import * as fs from 'fs/promises';
import * as crypto from 'crypto';
import { PathResolver } from '../utils/pathResolver';
import { Logger } from '../utils/logger';
```

```
import { DocumentRecord, ChunkRecord, PolicySection } from '../types/database';
import { FileService } from './fileService';

export class IndexService {
  private chroma: ChromaApi;
  private sqlite: Database.Database;
  private pathResolver: PathResolver;
  private logger: Logger;
  private fileService: FileService;
  private embeddingFunction: OpenAIEmbeddingFunction;

  constructor() {
    this.pathResolver = PathResolver.getInstance();
    this.logger = Logger.getInstance();
    this.fileService = new FileService();

    // Initialize Chroma client
    this.chroma = new ChromaApi({
      path: this.pathResolver.chromaDir
    });

    // Initialize embedding function
    this.embeddingFunction = new OpenAIEmbeddingFunction({
      openai_api_key: process.env.OPENAI_API_KEY || vscode.workspace.getConfiguration('safeAppeals').get('
      openai_model: "text-embedding-3-small"
    });

    // Initialize SQLite database
    this.sqlite = new Database(this.pathResolver.sqlitePath);
    this.initializeTables();
  }

  private initializeTables(): void {
    this.sqlite.exec(`
      PRAGMA foreign_keys = ON;

      CREATE TABLE IF NOT EXISTS documents (
        id TEXT PRIMARY KEY,
        filename TEXT NOT NULL,
        filepath TEXT NOT NULL,
        filetype TEXT NOT NULL,
        filesize INTEGER NOT NULL,
        uploadedAt DATETIME DEFAULT CURRENT_TIMESTAMP,
        lastIndexed DATETIME DEFAULT CURRENT_TIMESTAMP,
        checksum TEXT,
        metadata TEXT
      );

      CREATE TABLE IF NOT EXISTS chunks (
        chunkId TEXT PRIMARY KEY,
        docId TEXT NOT NULL,
        text TEXT NOT NULL,
        chunkIndex INTEGER NOT NULL,
        tokens INTEGER,
        FOREIGN KEY(docId) REFERENCES documents(id) ON DELETE CASCADE
      );

      CREATE TABLE IF NOT EXISTS policy_sections (
        sectionId TEXT PRIMARY KEY,
        title TEXT NOT NULL,
        level INTEGER NOT NULL,
        parentId TEXT,
        docId TEXT NOT NULL,
        pageNumber INTEGER,
        chunkIds TEXT, -- JSON array of chunk IDs
        FOREIGN KEY(docId) REFERENCES documents(id) ON DELETE CASCADE,
        FOREIGN KEY(parentId) REFERENCES policy_sections(sectionId)
      );

      CREATE TABLE IF NOT EXISTS workspace_configs (
        id TEXT PRIMARY KEY,
        name TEXT NOT NULL,
```

```
        rootPath TEXT NOT NULL,
        folderStructure TEXT NOT NULL,
        lastOrganized DATETIME,
        totalDocuments INTEGER DEFAULT 0,
        indexedDocuments INTEGER DEFAULT 0
      );

      CREATE TABLE IF NOT EXISTS search_history (
        id TEXT PRIMARY KEY,
        query TEXT NOT NULL,
        collection TEXT NOT NULL,
        timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
        resultCount INTEGER,
        responseTime INTEGER
      );

      -- Indexes for performance
      CREATE INDEX IF NOT EXISTS idx_documents_filetype ON documents(filetype);
      CREATE INDEX IF NOT EXISTS idx_documents_uploaded ON documents(uploadedAt);
      CREATE INDEX IF NOT EXISTS idx_chunks_docid ON chunks(docId);
      CREATE INDEX IF NOT EXISTS idx_policy_sections_docid ON policy_sections(docId);
      CREATE INDEX IF NOT EXISTS idx_policy_sections_parent ON policy_sections(parentId);
      CREATE INDEX IF NOT EXISTS idx_search_history_timestamp ON search_history(timestamp);
    `);
  }

  public async indexDocument(filePath: string, isPolicyManual: boolean = false): Promise&lt;string&gt; {
    const startTime = Date.now();

    try {
      // Generate document ID and checksum
      const fileBuffer = await fs.readFile(filePath);
      const checksum = crypto.createHash('sha256').update(fileBuffer).digest('hex');
      const docId = `${path.basename(filePath)}-${checksum.substring(0, 8)}`;

      // Check if document already indexed
      const existingDoc = this.sqlite.prepare(
        'SELECT id FROM documents WHERE checksum = ?'
      ).get(checksum);

      if (existingDoc) {
        this.logger.info(`Document ${filePath} already indexed with ID ${existingDoc.id}`);
        return existingDoc.id as string;
      }

      // Extract text content
      const { text, metadata } = await this.fileService.extractText(filePath);

      if (!text || text.trim().length === 0) {
        throw new Error('No text content extracted from document');
      }

      // Store document metadata in SQLite
      const stat = await fs.stat(filePath);
      const documentRecord: Omit&lt;DocumentRecord, 'uploadedAt' | 'lastIndexed'&gt; = {
        id: docId,
        filename: path.basename(filePath),
        filepath: filePath,
        filetype: path.extname(filePath).slice(1).toLowerCase(),
        filesize: stat.size,
        checksum,
        metadata: JSON.stringify(metadata)
      };

      this.sqlite.prepare(`
        INSERT INTO documents (id, filename, filepath, filetype, filesize, checksum, metadata)
        VALUES (?, ?, ?, ?, ?, ?, ?)
      `).run(
        documentRecord.id,
        documentRecord.filename,
        documentRecord.filepath,
        documentRecord.filetype,
```

```
      documentRecord.filesize,
      documentRecord.checksum,
      documentRecord.metadata
    );

    // Process and chunk text
    const chunks = await this.chunkText(text);

    // Store chunks in SQLite and prepare for Chroma
    const chunkRecords: ChunkRecord[] = [];
    const chromaDocuments: string[] = [];
    const chromaMetadatas: any[] = [];
    const chromaIds: string[] = [];

    const insertChunk = this.sqlite.prepare(`
      INSERT INTO chunks (chunkId, docId, text, chunkIndex, tokens)
      VALUES (?, ?, ?, ?, ?)
    `);

    for (let i = 0; i &lt; chunks.length; i++) {
      const chunkId = `${docId}-chunk-${i}`;
      const tokens = this.estimateTokens(chunks[i]);

      const chunkRecord: ChunkRecord = {
        chunkId,
        docId,
        text: chunks[i],
        chunkIndex: i,
        tokens
      };

      chunkRecords.push(chunkRecord);
      insertChunk.run(chunkId, docId, chunks[i], i, tokens);

      // Prepare for Chroma
      chromaDocuments.push(chunks[i]);
      chromaMetadatas.push({
        docId,
        chunkId,
        filename: documentRecord.filename,
        filetype: documentRecord.filetype,
        chunkIndex: i,
        isPolicyManual
      });
      chromaIds.push(chunkId);
    }

    // Store embeddings in Chroma
    const collectionName = isPolicyManual ? 'policy_manual' : 'workspace_docs';
    let collection: Collection;

    try {
      collection = await this.chroma.getCollection({
        name: collectionName,
        embeddingFunction: this.embeddingFunction
      });
    } catch {
      collection = await this.chroma.createCollection({
        name: collectionName,
        embeddingFunction: this.embeddingFunction
      });
    }

    await collection.add({
      ids: chromaIds,
      documents: chromaDocuments,
      metadatas: chromaMetadatas
    });

    // Process policy sections if it's a policy manual
    if (isPolicyManual) {
      await this.extractPolicySections(docId, text, chunkRecords);
```

```
      }

      const processingTime = Date.now() - startTime;
      this.logger.info(`Successfully indexed document ${filePath} in ${processingTime}ms with ${chunks.leng

      return docId;

    } catch (error) {
      this.logger.error(`Failed to index document ${filePath}:`, error);
      throw error;
    }
  }

  private async chunkText(text: string, chunkSize: number = 500, overlap: number = 50): Promise&amp;lt;string[]
    const sentences = text.split(/[.!?]+/).filter(s =&amp;gt; s.trim().length &amp;gt; 0);
    const chunks: string[] = [];
    let currentChunk = '';

    for (const sentence of sentences) {
      const trimmedSentence = sentence.trim();

      if ((currentChunk + ' ' + trimmedSentence).length &amp;gt; chunkSize &amp;amp;&amp;amp; currentChunk.length &amp;gt;
        chunks.push(currentChunk.trim());

        // Create overlap by including last few words
        const words = currentChunk.split(' ');
        const overlapWords = words.slice(-Math.min(overlap, words.length));
        currentChunk = overlapWords.join(' ') + ' ' + trimmedSentence;
      } else {
        currentChunk += (currentChunk ? ' ' : '') + trimmedSentence;
      }
    }

    if (currentChunk.trim().length &amp;gt; 0) {
      chunks.push(currentChunk.trim());
    }

    return chunks.filter(chunk =&amp;gt; chunk.length &amp;gt; 10); // Filter out very short chunks
  }

  private estimateTokens(text: string): number {
    // Rough approximation: 1 token ≈ 4 characters for English
    return Math.ceil(text.length / 4);
  }

  private async extractPolicySections(docId: string, text: string, chunks: ChunkRecord[]): Promise&amp;lt;void&
    // Extract headings and structure using regex patterns
    const headingPatterns = [
      /^(Chapter|Section|Article)\s+(\d+(?:\.\d+)*):?\s*(.+)$/gim,
      /^(\d+(?:\.\d+)*\.?)\s+(.+)$/gim,
      /^([A-Z][A-Z\s&amp;amp;-]+)$/gim
    ];

    const sections: PolicySection[] = [];
    const lines = text.split('\n');

    for (let lineIndex = 0; lineIndex &amp;lt; lines.length; lineIndex++) {
      const line = lines[lineIndex].trim();

      if (line.length === 0) continue;

      for (const pattern of headingPatterns) {
        pattern.lastIndex = 0; // Reset regex
        const match = pattern.exec(line);

        if (match) {
          const sectionId = `${docId}-section-${sections.length}`;
          const level = this.determineSectionLevel(match[1] || match[0]);
          const title = match[3] || match[2] || match[1];

          // Find associated chunks
          const associatedChunks = this.findChunksForSection(line, chunks);
```

```
        const section: PolicySection = {
          sectionId,
          title: title.trim(),
          level,
          docId,
          chunkIds: associatedChunks.map(c =&gt; c.chunkId)
        };

        sections.push(section);
        break;
      }
    }
  }

  // Insert sections into database
  const insertSection = this.sqlite.prepare(`
    INSERT INTO policy_sections (sectionId, title, level, docId, chunkIds)
    VALUES (?, ?, ?, ?, ?)
  `);

  for (const section of sections) {
    insertSection.run(
      section.sectionId,
      section.title,
      section.level,
      section.docId,
      JSON.stringify(section.chunkIds)
    );
  }

  this.logger.info(`Extracted ${sections.length} policy sections from document ${docId}`);
}

private determineSectionLevel(indicator: string): number {
  if (indicator.toLowerCase().includes('chapter')) return 1;
  if (indicator.toLowerCase().includes('section')) return 2;
  if (indicator.toLowerCase().includes('article')) return 2;
  if (/^\d+$/.test(indicator)) return 2;
  if (/^\d+\.\d+$/.test(indicator)) return 3;
  if (/^\d+\.\d+\.\d+$/.test(indicator)) return 4;
  return 3; // Default level
}

private findChunksForSection(sectionText: string, chunks: ChunkRecord[]): ChunkRecord[] {
  const relevantChunks: ChunkRecord[] = [];

  // Simple matching - in production, you might use more sophisticated matching
  for (const chunk of chunks) {
    if (chunk.text.toLowerCase().includes(sectionText.toLowerCase().substring(0, 50))) {
      relevantChunks.push(chunk);
    }
  }

  return relevantChunks;
}

public async searchSimilar(query: string, collectionName: 'policy_manual' | 'workspace_docs', limit: numb
  const startTime = Date.now();

  try {
    const collection = await this.chroma.getCollection({
      name: collectionName,
      embeddingFunction: this.embeddingFunction
    });

    const results = await collection.query({
      queryTexts: [query],
      nResults: limit
    });

    const responseTime = Date.now() - startTime;
```

```
      // Log search history
      this.sqlite.prepare(`
        INSERT INTO search_history (id, query, collection, resultCount, responseTime)
        VALUES (?, ?, ?, ?, ?)
      `).run(
        crypto.randomUUID(),
        query,
        collectionName,
        results.documents[0]?.length || 0,
        responseTime
      );

      return results.documents[0] || [];

    } catch (error) {
      this.logger.error(`Search failed for query "${query}" in collection ${collectionName}:`, error);
      throw error;
    }
  }

  public async getDocumentStats(): Promise&lt;any&gt; {
    const stats = this.sqlite.prepare(`
      SELECT
        COUNT(*) as totalDocuments,
        SUM(filesize) as totalSize,
        filetype,
        COUNT(*) as typeCount
      FROM documents
      GROUP BY filetype
    `).all();

    const chunkStats = this.sqlite.prepare(`
      SELECT COUNT(*) as totalChunks, AVG(tokens) as avgTokens
      FROM chunks
    `).get();

    return {
      documents: stats,
      chunks: chunkStats
    };
  }

  public async cleanup(): Promise&lt;void&gt; {
    try {
      this.sqlite.close();
      // Chroma cleanup if needed
    } catch (error) {
      this.logger.error('Error during cleanup:', error);
    }
  }
}
```

File Service for MIME Type Handling

src/services/fileService.ts

```
import * as fs from 'fs/promises';
import * as path from 'path';
import * as pdfjsLib from 'pdfjs-dist';
import * as mammoth from 'mammoth';
import { Logger } from '../utils/logger';

export interface ExtractedContent {
  text: string;
  metadata: {
    pageCount?: number;
    wordCount?: number;
    language?: string;
    author?: string;
```

```
    title?: string;
    createdDate?: Date;
    modifiedDate?: Date;
  };
}

export class FileService {
  private logger: Logger;

  constructor() {
    this.logger = Logger.getInstance();

    // Configure PDF.js worker
    if (typeof window === 'undefined') {
      // Node.js environment
      pdfjsLib.GlobalWorkerOptions.workerSrc = require.resolve('pdfjs-dist/build/pdf.worker.js');
    }
  }

  public async extractText(filePath: string): Promise&lt;ExtractedContent&gt; {
    const ext = path.extname(filePath).toLowerCase();
    const stat = await fs.stat(filePath);

    const baseMetadata = {
      createdDate: stat.birthtime,
      modifiedDate: stat.mtime
    };

    try {
      switch (ext) {
        case '.pdf':
          return await this.extractFromPdf(filePath, baseMetadata);
        case '.docx':
          return await this.extractFromDocx(filePath, baseMetadata);
        case '.doc':
          return await this.extractFromDoc(filePath, baseMetadata);
        case '.txt':
          return await this.extractFromText(filePath, baseMetadata);
        case '.md':
          return await this.extractFromMarkdown(filePath, baseMetadata);
        case '.rtf':
          return await this.extractFromRtf(filePath, baseMetadata);
        case '.odt':
          return await this.extractFromOdt(filePath, baseMetadata);
        default:
          throw new Error(`Unsupported file type: ${ext}`);
      }
    } catch (error) {
      this.logger.error(`Failed to extract text from ${filePath}:`, error);
      throw error;
    }
  }

  private async extractFromPdf(filePath: string, baseMetadata: any): Promise&lt;ExtractedContent&gt; {
    const buffer = await fs.readFile(filePath);
    const pdf = await pdfjsLib.getDocument({ data: buffer }).promise;

    let fullText = '';
    const pageTexts: string[] = [];

    for (let pageNum = 1; pageNum &lt;= pdf.numPages; pageNum++) {
      const page = await pdf.getPage(pageNum);
      const textContent = await page.getTextContent();

      const pageText = textContent.items
        .filter((item: any) =&gt; 'str' in item)
        .map((item: any) =&gt; item.str)
        .join(' ');

      pageTexts.push(pageText);
      fullText += pageText + '\n';
    }
```

```
    // Extract metadata from PDF
    const metadata = await pdf.getMetadata();

    return {
      text: fullText.trim(),
      metadata: {
        ...baseMetadata,
        pageCount: pdf.numPages,
        wordCount: this.countWords(fullText),
        title: metadata.info?.Title || path.basename(filePath, ext),
        author: metadata.info?.Author,
        language: this.detectLanguage(fullText)
      }
    };
  }

  private async extractFromDocx(filePath: string, baseMetadata: any): Promise&lt;ExtractedContent&gt; {
    const buffer = await fs.readFile(filePath);
    const result = await mammoth.extractRawText({ buffer });

    if (result.messages.length &gt; 0) {
      this.logger.warn(`DOCX extraction warnings for ${filePath}:`, result.messages);
    }

    return {
      text: result.value,
      metadata: {
        ...baseMetadata,
        wordCount: this.countWords(result.value),
        language: this.detectLanguage(result.value),
        title: path.basename(filePath, path.extname(filePath))
      }
    };
  }

  private async extractFromDoc(filePath: string, baseMetadata: any): Promise&lt;ExtractedContent&gt; {
    // For .doc files, we'd need a different library or converter
    // This is a placeholder implementation
    throw new Error('Legacy .doc format not yet supported. Please convert to .docx');
  }

  private async extractFromText(filePath: string, baseMetadata: any): Promise&lt;ExtractedContent&gt; {
    const text = await fs.readFile(filePath, 'utf-8');

    return {
      text,
      metadata: {
        ...baseMetadata,
        wordCount: this.countWords(text),
        language: this.detectLanguage(text),
        title: path.basename(filePath, path.extname(filePath))
      }
    };
  }

  private async extractFromMarkdown(filePath: string, baseMetadata: any): Promise&lt;ExtractedContent&gt; {
    const text = await fs.readFile(filePath, 'utf-8');

    // Extract title from markdown (first # heading)
    const titleMatch = text.match(/^#\s+(.+)$/m);
    const title = titleMatch ? titleMatch[1] : path.basename(filePath, path.extname(filePath));

    return {
      text,
      metadata: {
        ...baseMetadata,
        wordCount: this.countWords(text),
        language: this.detectLanguage(text),
        title
      }
    };
```

```
  }

  private async extractFromRtf(filePath: string, baseMetadata: any): Promise&lt;ExtractedContent&gt; {
    // RTF parsing would require additional library
    throw new Error('RTF format not yet supported');
  }

  private async extractFromOdt(filePath: string, baseMetadata: any): Promise&lt;ExtractedContent&gt; {
    // ODT parsing would require additional library
    throw new Error('ODT format not yet supported');
  }

  private countWords(text: string): number {
    return text.trim().split(/\s+/).filter(word =&gt; word.length &gt; 0).length;
  }

  private detectLanguage(text: string): string {
    // Simple language detection - in production, use a proper language detection library
    const sample = text.substring(0, 1000).toLowerCase();

    // English indicators
    const englishWords = ['the', 'and', 'for', 'are', 'but', 'not', 'you', 'all', 'can', 'had', 'her', 'was
    const englishCount = englishWords.reduce((count, word) =&gt;
      count + (sample.split(word).length - 1), 0
    );

    if (englishCount &gt; 10) return 'en';

    // Add more language detection as needed
    return 'unknown';
  }

  public getSupportedExtensions(): string[] {
    return ['.pdf', '.docx', '.txt', '.md'];
  }

  public isSupported(filePath: string): boolean {
    const ext = path.extname(filePath).toLowerCase();
    return this.getSupportedExtensions().includes(ext);
  }
}
```

Agent Service & Tool Calls

Comprehensive Agent Service

src/services/agentService.ts

```
import * as vscode from 'vscode';
import { OpenAI } from 'openai';
import { IndexService } from './indexService';
import { Logger } from '../utils/logger';
import { PathResolver } from '../utils/pathResolver';
import * as fs from 'fs/promises';
import * as path from 'path';

export interface AgentTool {
  name: string;
  description: string;
  parameters: any;
  handler: (params: any) =&gt; Promise&lt;any&gt;;
}

export class AgentService {
  private openai: OpenAI;
  private indexService: IndexService;
  private logger: Logger;
  private pathResolver: PathResolver;
  private tools: Map&lt;string, AgentTool&gt; = new Map();
```

```
constructor() {
  this.openai = new OpenAI({
    apiKey: process.env.OPENAI_API_KEY || vscode.workspace.getConfiguration('safeAppeals').get('openaiAp
  });
  this.indexService = new IndexService();
  this.logger = Logger.getInstance();
  this.pathResolver = PathResolver.getInstance();

  this.registerTools();
}

private registerTools(): void {
  // Tool: Upload and index policy manual
  this.tools.set('upload_policy_manual', {
    name: 'upload_policy_manual',
    description: 'Upload and index a policy manual document for RAG queries',
    parameters: {
      type: 'object',
      properties: {
        filePath: {
          type: 'string',
          description: 'Path to the policy manual file'
        }
      },
      required: ['filePath']
    },
    handler: this.uploadPolicyManual.bind(this)
  });

  // Tool: Search policy manual
  this.tools.set('search_policy', {
    name: 'search_policy',
    description: 'Search the policy manual for specific information',
    parameters: {
      type: 'object',
      properties: {
        query: {
          type: 'string',
          description: 'The search query for policy information'
        },
        limit: {
          type: 'number',
          description: 'Maximum number of results to return',
          default: 5
        }
      },
      required: ['query']
    },
    handler: this.searchPolicy.bind(this)
  });

  // Tool: Index workspace documents
  this.tools.set('index_workspace', {
    name: 'index_workspace',
    description: 'Index all documents in a workspace folder',
    parameters: {
      type: 'object',
      properties: {
        folderPath: {
          type: 'string',
          description: 'Path to the folder containing documents to index'
        }
      },
      required: ['folderPath']
    },
    handler: this.indexWorkspace.bind(this)
  });

  // Tool: Organize workspace files
  this.tools.set('organize_workspace', {
    name: 'organize_workspace',
```

```
      description: 'Automatically organize workspace files into structured folders based on content',
      parameters: {
        type: 'object',
        properties: {
          sourcePath: {
            type: 'string',
            description: 'Path to the folder containing files to organize'
          },
          structureTemplate: {
            type: 'string',
            description: 'Name of the folder structure template to use',
            default: 'workers_compensation'
          }
        },
        required: ['sourcePath']
      },
      handler: this.organizeWorkspace.bind(this)
    });

    // Tool: Search workspace documents
    this.tools.set('search_workspace', {
      name: 'search_workspace',
      description: 'Search indexed workspace documents',
      parameters: {
        type: 'object',
        properties: {
          query: {
            type: 'string',
            description: 'The search query for workspace documents'
          },
          limit: {
            type: 'number',
            description: 'Maximum number of results to return',
            default: 5
          }
        },
        required: ['query']
      },
      handler: this.searchWorkspace.bind(this)
    });

    // Tool: Generate document summary
    this.tools.set('summarize_document', {
      name: 'summarize_document',
      description: 'Generate a summary of a specific document',
      parameters: {
        type: 'object',
        properties: {
          documentId: {
            type: 'string',
            description: 'ID of the document to summarize'
          }
        },
        required: ['documentId']
      },
      handler: this.summarizeDocument.bind(this)
    });
  }

  private async uploadPolicyManual(params: { filePath: string }): Promise&lt;any&gt; {
    try {
      this.logger.info(`Uploading policy manual: ${params.filePath}`);

      const docId = await this.indexService.indexDocument(params.filePath, true);

      return {
        success: true,
        documentId: docId,
        message: `Policy manual successfully indexed with ID: ${docId}`
      };
    } catch (error) {
      this.logger.error('Failed to upload policy manual:', error);
```

```
      return {
        success: false,
        error: error instanceof Error ? error.message : 'Unknown error occurred'
      };
    }
  }

  private async searchPolicy(params: { query: string; limit?: number }): Promise&lt;any&gt; {
    try {
      const results = await this.indexService.searchSimilar(
        params.query,
        'policy_manual',
        params.limit || 5
      );

      if (results.length === 0) {
        return {
          success: true,
          results: [],
          message: 'No relevant policy information found for the query.'
        };
      }

      // Generate contextual response using found information
      const context = results.join('\n\n');
      const response = await this.generateContextualResponse(params.query, context, 'policy');

      return {
        success: true,
        results: results,
        answer: response,
        resultCount: results.length
      };
    } catch (error) {
      this.logger.error('Policy search failed:', error);
      return {
        success: false,
        error: error instanceof Error ? error.message : 'Search failed'
      };
    }
  }

  private async indexWorkspace(params: { folderPath: string }): Promise&lt;any&gt; {
    try {
      const files = await fs.readdir(params.folderPath, { withFileTypes: true });
      const documentFiles = files
        .filter(file =&gt; file.isFile())
        .map(file =&gt; path.join(params.folderPath, file.name))
        .filter(filePath =&gt; {
          const ext = path.extname(filePath).toLowerCase();
          return ['.pdf', '.docx', '.txt', '.md'].includes(ext);
        });

      const results = {
        indexed: 0,
        failed: 0,
        documentIds: [] as string[]
      };

      for (const filePath of documentFiles) {
        try {
          const docId = await this.indexService.indexDocument(filePath, false);
          results.indexed++;
          results.documentIds.push(docId);
          this.logger.info(`Indexed: ${filePath}`);
        } catch (error) {
          results.failed++;
          this.logger.error(`Failed to index ${filePath}:`, error);
        }
      }

      return {
```

```
      success: true,
      ...results,
      message: `Indexed ${results.indexed} documents, ${results.failed} failed`
    };
  } catch (error) {
    this.logger.error('Workspace indexing failed:', error);
    return {
      success: false,
      error: error instanceof Error ? error.message : 'Indexing failed'
    };
  }
}

private async organizeWorkspace(params: { sourcePath: string; structureTemplate?: string }): Promise&lt;a
  try {
    // Load folder structure template
    const templatePath = path.join(
      __dirname,
      '..',
      '..',
      'resources',
      'templates',
      `${params.structureTemplate || 'workers_compensation'}.json`
    );

    const structureTemplate = JSON.parse(await fs.readFile(templatePath, 'utf-8'));

    // Get list of files to organize
    const files = await fs.readdir(params.sourcePath, { withFileTypes: true });
    const filesToOrganize = files
      .filter(file =&gt; file.isFile())
      .map(file =&gt; path.join(params.sourcePath, file.name));

    const organizationResults = {
      organized: 0,
      failed: 0,
      folderCounts: {} as Record&lt;string, number&gt;
    };

    for (const filePath of filesToOrganize) {
      try {
        const category = await this.classifyDocument(filePath, structureTemplate);
        const targetFolder = path.join(path.dirname(params.sourcePath), category);

        // Create target folder if it doesn't exist
        await fs.mkdir(targetFolder, { recursive: true });

        // Move file
        const fileName = path.basename(filePath);
        const targetPath = path.join(targetFolder, fileName);
        await fs.rename(filePath, targetPath);

        organizationResults.organized++;
        organizationResults.folderCounts[category] = (organizationResults.folderCounts[category] || 0) +

        this.logger.info(`Moved ${fileName} to ${category}`);
      } catch (error) {
        organizationResults.failed++;
        this.logger.error(`Failed to organize ${filePath}:`, error);
      }
    }

    return {
      success: true,
      ...organizationResults,
      message: `Organized ${organizationResults.organized} files into structured folders`
    };
  } catch (error) {
    this.logger.error('Workspace organization failed:', error);
    return {
      success: false,
      error: error instanceof Error ? error.message : 'Organization failed'
```

```
      };
    }
  }

  private async classifyDocument(filePath: string, structureTemplate: any): Promise&lt;string&gt; {
    try {
      // Extract a sample of text from the document
      const { text } = await new (await import('./fileService')).FileService().extractText(filePath);
      const sample = text.substring(0, 2000); // First 2000 characters

      // Use AI to classify the document
      const prompt = `
Analyze this document sample and classify it into one of these categories:
${JSON.stringify(Object.keys(structureTemplate), null, 2)}

Document sample:
"${sample}"

Based on the content, which category best fits this document? Respond with only the category name from the
`;

      const response = await this.openai.chat.completions.create({
        model: 'gpt-3.5-turbo',
        messages: [
          {
            role: 'system',
            content: 'You are a document classifier for workers compensation case management. Classify docu
          },
          {
            role: 'user',
            content: prompt
          }
        ],
        temperature: 0.1,
        max_tokens: 50
      });

      const classification = response.choices[0]?.message?.content?.trim();

      // Validate classification against template
      if (classification &amp;&amp; Object.keys(structureTemplate).includes(classification)) {
        return classification;
      }

      // Fallback to pattern matching if AI classification fails
      return this.fallbackClassification(filePath, sample, structureTemplate);

    } catch (error) {
      this.logger.error(`Failed to classify document ${filePath}:`, error);
      return 'Miscellaneous'; // Default folder
    }
  }

  private fallbackClassification(filePath: string, content: string, template: any): string {
    const fileName = path.basename(filePath).toLowerCase();
    const contentLower = content.toLowerCase();

    // Pattern-based classification
    const patterns = {
      'Medical Records': ['medical', 'doctor', 'physician', 'diagnosis', 'treatment', 'hospital', 'clinic']
      'Legal Documents': ['legal', 'court', 'judgment', 'appeal', 'hearing', 'attorney', 'lawyer'],
      'Correspondence': ['letter', 'email', 'correspondence', 'communication', 'reply'],
      'Policy Documents': ['policy', 'regulation', 'rule', 'guideline', 'procedure'],
      'Financial Records': ['payment', 'invoice', 'receipt', 'financial', 'cost', 'bill']
    };

    for (const [category, keywords] of Object.entries(patterns)) {
      if (Object.keys(template).includes(category)) {
        const matches = keywords.some(keyword =&gt;
          fileName.includes(keyword) || contentLower.includes(keyword)
        );
        if (matches) return category;
```

```
      }
    }

    return 'Miscellaneous';
  }

  private async searchWorkspace(params: { query: string; limit?: number }): Promise&lt;any&gt; {
    try {
      const results = await this.indexService.searchSimilar(
        params.query,
        'workspace_docs',
        params.limit || 5
      );

      if (results.length === 0) {
        return {
          success: true,
          results: [],
          message: 'No relevant documents found in workspace.'
        };
      }

      const context = results.join('\n\n');
      const response = await this.generateContextualResponse(params.query, context, 'workspace');

      return {
        success: true,
        results: results,
        answer: response,
        resultCount: results.length
      };
    } catch (error) {
      this.logger.error('Workspace search failed:', error);
      return {
        success: false,
        error: error instanceof Error ? error.message : 'Search failed'
      };
    }
  }

  private async summarizeDocument(params: { documentId: string }): Promise&lt;any&gt; {
    // Implementation for document summarization
    return {
      success: true,
      summary: 'Document summary functionality to be implemented',
      documentId: params.documentId
    };
  }

  private async generateContextualResponse(query: string, context: string, type: 'policy' | 'workspace'): P
    const systemPrompt = type === 'policy'
      ? 'You are an expert in workers compensation policy. Provide accurate, helpful answers based on polic
      : 'You are a document analysis assistant. Help users understand their workspace documents.';

    const response = await this.openai.chat.completions.create({
      model: 'gpt-4',
      messages: [
        {
          role: 'system',
          content: systemPrompt
        },
        {
          role: 'user',
          content: `Based on this context:\n\n${context}\n\nPlease answer: ${query}`
        }
      ],
      temperature: 0.2,
      max_tokens: 500
    });

    return response.choices[0]?.message?.content || 'No response generated';
  }
```

```
  public async executeAgentCall(toolName: string, parameters: any): Promise&lt;any&gt; {
    const tool = this.tools.get(toolName);

    if (!tool) {
      throw new Error(`Unknown tool: ${toolName}`);
    }

    this.logger.info(`Executing agent tool: ${toolName}`, parameters);

    try {
      const result = await tool.handler(parameters);
      this.logger.info(`Agent tool ${toolName} completed successfully`);
      return result;
    } catch (error) {
      this.logger.error(`Agent tool ${toolName} failed:`, error);
      throw error;
    }
  }

  public getAvailableTools(): AgentTool[] {
    return Array.from(this.tools.values());
  }

  public async cleanup(): Promise&lt;void&gt; {
    await this.indexService.cleanup();
  }
}
```

Policy Manual Dashboard

Dashboard Provider Implementation

src/dashboard/policyDashboard.ts

```
import * as vscode from 'vscode';
import { IndexService } from '../services/indexService';
import { AgentService } from '../services/agentService';
import { Logger } from '../utils/logger';
import * as path from 'path';

export class PolicyDashboardProvider implements vscode.WebviewViewProvider {
  public static readonly viewType = 'safeAppeals.policyDashboard';

  private _view?: vscode.WebviewView;
  private indexService: IndexService;
  private agentService: AgentService;
  private logger: Logger;

  constructor(private readonly _extensionUri: vscode.Uri) {
    this.indexService = new IndexService();
    this.agentService = new AgentService();
    this.logger = Logger.getInstance();
  }

  public resolveWebviewView(
    webviewView: vscode.WebviewView,
    context: vscode.WebviewViewResolveContext,
    _token: vscode.CancellationToken,
  ) {
    this._view = webviewView;

    webviewView.webview.options = {
      enableScripts: true,
      localResourceRoots: [this._extensionUri]
    };

    webviewView.webview.html = this._getHtmlForWebview(webviewView.webview);
```

```
  webviewView.webview.onDidReceiveMessage(async data =&gt; {
    switch (data.type) {
      case 'uploadPolicy':
        await this.handleUploadPolicy();
        break;
      case 'searchPolicy':
        await this.handleSearchPolicy(data.query);
        break;
      case 'loadPolicySections':
        await this.handleLoadPolicySections();
        break;
      case 'viewSection':
        await this.handleViewSection(data.sectionId);
        break;
    }
  });

  // Load initial data
  this.loadInitialData();
}

private async handleUploadPolicy(): Promise&lt;void&gt; {
  try {
    const fileUri = await vscode.window.showOpenDialog({
      canSelectFiles: true,
      canSelectFolders: false,
      canSelectMany: false,
      filters: {
        'PDF Files': ['pdf'],
        'Word Documents': ['docx'],
        'Text Files': ['txt'],
        'All Supported': ['pdf', 'docx', 'txt', 'md']
      },
      openLabel: 'Upload Policy Manual'
    });

    if (fileUri &amp;&amp; fileUri[0]) {
      const result = await this.agentService.executeAgentCall('upload_policy_manual', {
        filePath: fileUri[0].fsPath
      });

      this._view?.webview.postMessage({
        type: 'uploadResult',
        success: result.success,
        message: result.message,
        documentId: result.documentId
      });

      if (result.success) {
        await this.loadPolicySections();
      }
    }
  } catch (error) {
    this.logger.error('Policy upload failed:', error);
    this._view?.webview.postMessage({
      type: 'uploadResult',
      success: false,
      message: 'Upload failed: ' + (error instanceof Error ? error.message : 'Unknown error')
    });
  }
}

private async handleSearchPolicy(query: string): Promise&lt;void&gt; {
  try {
    const result = await this.agentService.executeAgentCall('search_policy', {
      query: query,
      limit: 10
    });

    this._view?.webview.postMessage({
      type: 'searchResults',
      success: result.success,
```

```
      results: result.results,
      answer: result.answer,
      query: query
    });
  } catch (error) {
    this.logger.error('Policy search failed:', error);
    this._view?.webview.postMessage({
      type: 'searchResults',
      success: false,
      error: error instanceof Error ? error.message : 'Search failed'
    });
  }
}

private async handleLoadPolicySections(): Promise&lt;void&gt; {
  await this.loadPolicySections();
}

private async handleViewSection(sectionId: string): Promise&lt;void&gt; {
  // Implementation to view specific policy section
  // This would fetch the section content and display it
}

private async loadInitialData(): Promise&lt;void&gt; {
  try {
    const stats = await this.indexService.getDocumentStats();
    this._view?.webview.postMessage({
      type: 'initialData',
      stats: stats
    });

    await this.loadPolicySections();
  } catch (error) {
    this.logger.error('Failed to load initial data:', error);
  }
}

private async loadPolicySections(): Promise&lt;void&gt; {
  try {
    // This would query the SQLite database for policy sections
    // Implementation depends on your specific database schema
    this._view?.webview.postMessage({
      type: 'policySections',
      sections: [] // Placeholder
    });
  } catch (error) {
    this.logger.error('Failed to load policy sections:', error);
  }
}

private _getHtmlForWebview(webview: vscode.Webview): string {
  // Get the local path to main script run in the webview
  const scriptPathOnDisk = vscode.Uri.joinPath(this._extensionUri, 'resources', 'webview', 'policy-dashbo
  const scriptUri = webview.asWebviewUri(scriptPathOnDisk);

  // Get path to stylesheet
  const stylePathOnDisk = vscode.Uri.joinPath(this._extensionUri, 'resources', 'webview', 'policy-dashboa
  const styleUri = webview.asWebviewUri(stylePathOnDisk);

  // Use a nonce to only allow specific scripts to be run
  const nonce = getNonce();

  return `
  &lt;html lang="en"&gt;
  &lt;head&gt;
    &lt;meta charset="UTF-8"&gt;
    &lt;meta http-equiv="Content-Security-Policy" content="default-src 'none'; style-src ${webview.cspSou
    &lt;meta name="viewport" content="width=device-width, initial-scale=1.0"&gt;
    &lt;link href="${styleUri}" rel="stylesheet"&gt;
    &lt;title&gt;Policy Manual Dashboard&lt;/title&gt;
  &lt;/head&gt;
  &lt;body&gt;
```

```
      <div>
        &lt;header&gt;
          <h2>Policy Manual</h2>
          &lt;button id="uploadBtn" class="primary-btn"&gt;Upload Policy Manual&lt;/button&gt;
        &lt;/header&gt;

        &lt;section class="search-section"&gt;
          <div>
            &lt;input type="text" id="searchInput" placeholder="Search policy manual..." /&gt;
            &lt;button id="searchBtn"&gt;Search&lt;/button&gt;
          </div>
          <div></div>
        &lt;/section&gt;

        &lt;section class="navigation-section"&gt;
          <h3>Policy Sections</h3>
          <div>
            <div>Loading policy structure...</div>
          </div>
        &lt;/section&gt;

        &lt;section class="stats-section"&gt;
          <h3>Statistics</h3>
          <div>
            <div>
              <span>Total Documents</span>
              <span>-</span>
            </div>
            <div>
              <span>Policy Sections</span>
              <span>-</span>
            </div>
            <div>
              <span>Last Updated</span>
              <span>-</span>
            </div>
          </div>
        &lt;/section&gt;
      </div>

      &lt;script nonce="${nonce}" src="${scriptUri}"&gt;&lt;/script&gt;
    &lt;/body&gt;
    &lt;/html&gt;`;
  }
}

function getNonce() {
  let text = '';
  const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
  for (let i = 0; i &lt; 32; i++) {
    text += possible.charAt(Math.floor(Math.random() * possible.length));
  }
  return text;
}
```

Dashboard JavaScript

resources/webview/policy-dashboard.js

```
(function() {
  const vscode = acquireVsCodeApi();

  // DOM elements
  const uploadBtn = document.getElementById('uploadBtn');
  const searchInput = document.getElementById('searchInput');
  const searchBtn = document.getElementById('searchBtn');
  const searchResults = document.getElementById('searchResults');
  const policyTree = document.getElementById('policyTree');
  const totalDocs = document.getElementById('totalDocs');
  const totalSections = document.getElementById('totalSections');
```

```
const lastUpdated = document.getElementById('lastUpdated');

// Event listeners
uploadBtn.addEventListener('click', () =&gt; {
  vscode.postMessage({ type: 'uploadPolicy' });
});

searchBtn.addEventListener('click', () =&gt; {
  performSearch();
});

searchInput.addEventListener('keypress', (e) =&gt; {
  if (e.key === 'Enter') {
    performSearch();
  }
});

function performSearch() {
  const query = searchInput.value.trim();
  if (!query) return;

  searchBtn.disabled = true;
  searchBtn.textContent = 'Searching...';

  vscode.postMessage({
    type: 'searchPolicy',
    query: query
  });
}

// Handle messages from extension
window.addEventListener('message', event =&gt; {
  const message = event.data;

  switch (message.type) {
    case 'uploadResult':
      handleUploadResult(message);
      break;
    case 'searchResults':
      handleSearchResults(message);
      break;
    case 'policySections':
      handlePolicySections(message);
      break;
    case 'initialData':
      handleInitialData(message);
      break;
  }
});

function handleUploadResult(message) {
  if (message.success) {
    showNotification('Policy manual uploaded successfully!', 'success');
    // Reload policy sections
    vscode.postMessage({ type: 'loadPolicySections' });
  } else {
    showNotification(`Upload failed: ${message.message}`, 'error');
  }
}

function handleSearchResults(message) {
  searchBtn.disabled = false;
  searchBtn.textContent = 'Search';

  if (message.success) {
    displaySearchResults(message.results, message.answer, message.query);
  } else {
    showNotification(`Search failed: ${message.error}`, 'error');
  }
}

function handlePolicySections(message) {
```

```
  renderPolicyTree(message.sections);
}

function handleInitialData(message) {
  updateStats(message.stats);
}

function displaySearchResults(results, answer, query) {
  searchResults.innerHTML = '';
  searchResults.classList.remove('hidden');

  if (results.length === 0) {
    searchResults.innerHTML = `
      <div>
        <p>No results found for "${query}"</p>
      </div>
    `;
    return;
  }

  const resultsHtml = `
    <div>
      <h4>Answer:</h4>
      <p>${answer}</p>
    </div>
    <div>
      <h4>Sources (${results.length}):</h4>
      ${results.map((result, index) =&gt; `
        <div>
          <div>
            ${result.substring(0, 200)}${result.length &gt; 200 ? '...' : ''}
          </div>
        </div>
      `).join('')}
    </div>
  `;

  searchResults.innerHTML = resultsHtml;
}

function renderPolicyTree(sections) {
  if (!sections || sections.length === 0) {
    policyTree.innerHTML = '<div>No policy sections available</div>';
    return;
  }

  // Build hierarchical tree structure
  const treeHtml = sections.map(section =&gt; `
    <div>
      <span></span>
      <span>${section.title}</span>
    </div>
  `).join('');

  policyTree.innerHTML = treeHtml;

  // Add click handlers
  policyTree.querySelectorAll('.tree-item').forEach(item =&gt; {
    item.addEventListener('click', (e) =&gt; {
      const sectionId = e.currentTarget.dataset.sectionId;
      vscode.postMessage({
        type: 'viewSection',
        sectionId: sectionId
      });
    });
  });
}

function updateStats(stats) {
  if (stats.documents) {
    const totalDocCount = stats.documents.reduce((sum, doc) =&gt; sum + doc.typeCount, 0);
    totalDocs.textContent = totalDocCount.toString();
```

```
    }

    lastUpdated.textContent = new Date().toLocaleDateString();
  }

  function showNotification(message, type) {
    // Create notification element
    const notification = document.createElement('div');
    notification.className = `notification ${type}`;
    notification.textContent = message;

    document.body.appendChild(notification);

    // Auto-remove after 5 seconds
    setTimeout(() =&gt; {
      if (notification.parentNode) {
        notification.parentNode.removeChild(notification);
      }
    }, 5000);
  }

  // Initialize
  vscode.postMessage({ type: 'loadPolicySections' });
})();
```

Dashboard CSS

resources/webview/policy-dashboard.css

```
:root {
  --primary-color: #A6E22E;
  --primary-hover: #8CBF22;
  --text-color: var(--vscode-foreground);
  --bg-color: var(--vscode-editor-background);
  --border-color: var(--vscode-panel-border);
  --secondary-bg: var(--vscode-panel-background);
}

* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}

body {
  font-family: var(--vscode-font-family);
  color: var(--text-color);
  background-color: var(--bg-color);
  line-height: 1.5;
}

.dashboard-container {
  padding: 16px;
  max-width: 100%;
  overflow-x: hidden;
}

header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-bottom: 24px;
  padding-bottom: 16px;
  border-bottom: 1px solid var(--border-color);
}

header h2 {
  color: var(--primary-color);
  font-size: 1.5rem;
  font-weight: 600;
```

```
}

.primary-btn {
  background-color: var(--primary-color);
  color: #000;
  border: none;
  padding: 8px 16px;
  border-radius: 4px;
  cursor: pointer;
  font-weight: 500;
  transition: background-color 0.2s;
}

.primary-btn:hover {
  background-color: var(--primary-hover);
}

.primary-btn:disabled {
  opacity: 0.6;
  cursor: not-allowed;
}

/* Search Section */
.search-section {
  margin-bottom: 24px;
}

.search-container {
  display: flex;
  gap: 8px;
  margin-bottom: 16px;
}

#searchInput {
  flex: 1;
  padding: 8px 12px;
  border: 1px solid var(--border-color);
  border-radius: 4px;
  background-color: var(--bg-color);
  color: var(--text-color);
  font-size: 14px;
}

#searchInput:focus {
  outline: none;
  border-color: var(--primary-color);
}

#searchBtn {
  background-color: var(--secondary-bg);
  color: var(--text-color);
  border: 1px solid var(--border-color);
  padding: 8px 16px;
  border-radius: 4px;
  cursor: pointer;
  transition: background-color 0.2s;
}

#searchBtn:hover {
  background-color: var(--primary-color);
  color: #000;
}

.search-results {
  border: 1px solid var(--border-color);
  border-radius: 4px;
  padding: 16px;
  background-color: var(--secondary-bg);
}

.search-results.hidden {
  display: none;
```

```
}

.search-answer {
  margin-bottom: 16px;
  padding-bottom: 16px;
  border-bottom: 1px solid var(--border-color);
}

.search-answer h4 {
  color: var(--primary-color);
  margin-bottom: 8px;
}

.search-sources h4 {
  color: var(--text-color);
  margin-bottom: 12px;
}

.search-result-item {
  margin-bottom: 12px;
  padding: 8px;
  background-color: var(--bg-color);
  border-radius: 4px;
  border-left: 3px solid var(--primary-color);
}

.result-snippet {
  font-size: 13px;
  line-height: 1.4;
}

.no-results {
  text-align: center;
  color: var(--vscode-descriptionForeground);
  padding: 24px;
}

/* Navigation Section */
.navigation-section {
  margin-bottom: 24px;
}

.navigation-section h3 {
  margin-bottom: 12px;
  color: var(--text-color);
}

.policy-tree {
  border: 1px solid var(--border-color);
  border-radius: 4px;
  background-color: var(--secondary-bg);
  max-height: 300px;
  overflow-y: auto;
}

.tree-item {
  display: flex;
  align-items: center;
  padding: 8px 12px;
  cursor: pointer;
  border-bottom: 1px solid var(--border-color);
  transition: background-color 0.2s;
}

.tree-item:hover {
  background-color: var(--bg-color);
}

.tree-item.level-1 {
  font-weight: 600;
  background-color: var(--bg-color);
}
```

```
.tree-item.level-2 {
  padding-left: 24px;
}

.tree-item.level-3 {
  padding-left: 36px;
}

.tree-item.level-4 {
  padding-left: 48px;
}

.tree-icon {
  margin-right: 8px;
  font-size: 14px;
}

.tree-title {
  font-size: 14px;
}

.empty-state, .loading {
  text-align: center;
  padding: 24px;
  color: var(--vscode-descriptionForeground);
}

/* Statistics Section */
.stats-section h3 {
  margin-bottom: 12px;
  color: var(--text-color);
}

.stats-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(120px, 1fr));
  gap: 12px;
}

.stat-item {
  display: flex;
  flex-direction: column;
  padding: 12px;
  background-color: var(--secondary-bg);
  border: 1px solid var(--border-color);
  border-radius: 4px;
  text-align: center;
}

.stat-label {
  font-size: 12px;
  color: var(--vscode-descriptionForeground);
  margin-bottom: 4px;
}

.stat-value {
  font-size: 18px;
  font-weight: 600;
  color: var(--primary-color);
}

/* Notifications */
.notification {
  position: fixed;
  top: 16px;
  right: 16px;
  padding: 12px 16px;
  border-radius: 4px;
  font-weight: 500;
  z-index: 1000;
  animation: slideIn 0.3s ease-out;
```

```
}

.notification.success {
  background-color: #4CAF50;
  color: white;
}

.notification.error {
  background-color: #f44336;
  color: white;
}

@keyframes slideIn {
  from {
    transform: translateX(100%);
    opacity: 0;
  }
  to {
    transform: translateX(0);
    opacity: 1;
  }
}

/* Responsive Design */
@media (max-width: 480px) {
  .dashboard-container {
    padding: 12px;
  }

  header {
    flex-direction: column;
    gap: 12px;
    align-items: stretch;
  }

  .search-container {
    flex-direction: column;
  }

  .stats-grid {
    grid-template-columns: 1fr;
  }
}
```

Extension Entry Point

Main Extension File

src/extension.ts

```
import * as vscode from 'vscode';
import { PolicyDashboardProvider } from './dashboard/policyDashboard';
import { AgentService } from './services/agentService';
import { PathResolver } from './utils/pathResolver';
import { Logger } from './utils/logger';

export async function activate(context: vscode.ExtensionContext) {
  const logger = Logger.getInstance();
  logger.info('Activating Safe Appeals Navigator extension');

  try {
    // Initialize path resolver and ensure directories exist
    const pathResolver = PathResolver.getInstance();
    await pathResolver.ensureDirectoriesExist();

    // Initialize services
    const agentService = new AgentService();

    // Register policy dashboard provider
```

```
    const policyDashboardProvider = new PolicyDashboardProvider(context.extensionUri);
    context.subscriptions.push(
      vscode.window.registerWebviewViewProvider(
        PolicyDashboardProvider.viewType,
        policyDashboardProvider
      )
    );

    // Register commands
    registerCommands(context, agentService);

    logger.info('Safe Appeals Navigator extension activated successfully');
  } catch (error) {
    logger.error('Failed to activate extension:', error);
    vscode.window.showErrorMessage('Failed to activate Safe Appeals Navigator: ' + error);
  }
}

function registerCommands(context: vscode.ExtensionContext, agentService: AgentService) {
  // Upload policy manual command
  const uploadPolicyCmd = vscode.commands.registerCommand(
    'safeAppeals.uploadPolicy',
    async () =&gt; {
      try {
        const fileUri = await vscode.window.showOpenDialog({
          canSelectFiles: true,
          canSelectFolders: false,
          canSelectMany: false,
          filters: {
            'Policy Documents': ['pdf', 'docx', 'txt', 'md']
          },
          openLabel: 'Select Policy Manual'
        });

        if (fileUri &amp;&amp; fileUri[0]) {
          await vscode.window.withProgress(
            {
              location: vscode.ProgressLocation.Notification,
              title: 'Uploading Policy Manual',
              cancellable: false
            },
            async (progress) =&gt; {
              progress.report({ message: 'Processing document...' });

              const result = await agentService.executeAgentCall('upload_policy_manual', {
                filePath: fileUri[0].fsPath
              });

              if (result.success) {
                vscode.window.showInformationMessage('Policy manual uploaded successfully!');
              } else {
                vscode.window.showErrorMessage(`Upload failed: ${result.error}`);
              }
            }
          );
        }
      } catch (error) {
        vscode.window.showErrorMessage('Failed to upload policy manual: ' + error);
      }
    }
  );

  // Index workspace command
  const indexWorkspaceCmd = vscode.commands.registerCommand(
    'safeAppeals.indexWorkspace',
    async () =&gt; {
      try {
        const folderUri = await vscode.window.showOpenDialog({
          canSelectFiles: false,
          canSelectFolders: true,
          canSelectMany: false,
          openLabel: 'Select Folder to Index'
```

```
      });

      if (folderUri &amp;amp;&amp;amp; folderUri[0]) {
        await vscode.window.withProgress(
          {
            location: vscode.ProgressLocation.Notification,
            title: 'Indexing Workspace',
            cancellable: false
          },
          async (progress) =&gt; {
            progress.report({ message: 'Analyzing documents...' });

            const result = await agentService.executeAgentCall('index_workspace', {
              folderPath: folderUri[0].fsPath
            });

            vscode.window.showInformationMessage(result.message);
          }
        );
      }
    } catch (error) {
      vscode.window.showErrorMessage('Failed to index workspace: ' + error);
    }
  }
);

// Organize workspace command
const organizeWorkspaceCmd = vscode.commands.registerCommand(
  'safeAppeals.organizeWorkspace',
  async () =&gt; {
    try {
      const folderUri = await vscode.window.showOpenDialog({
        canSelectFiles: false,
        canSelectFolders: true,
        canSelectMany: false,
        openLabel: 'Select Folder to Organize'
      });

      if (folderUri &amp;amp;&amp;amp; folderUri[0]) {
        const confirm = await vscode.window.showWarningMessage(
          'This will move files into organized folders. Continue?',
          'Yes',
          'Cancel'
        );

        if (confirm === 'Yes') {
          await vscode.window.withProgress(
            {
              location: vscode.ProgressLocation.Notification,
              title: 'Organizing Workspace',
              cancellable: false
            },
            async (progress) =&gt; {
              progress.report({ message: 'Classifying and moving files...' });

              const result = await agentService.executeAgentCall('organize_workspace', {
                sourcePath: folderUri[0].fsPath
              });

              vscode.window.showInformationMessage(result.message);
            }
          );
        }
      }
    } catch (error) {
      vscode.window.showErrorMessage('Failed to organize workspace: ' + error);
    }
  }
);

// Search policy command
const searchPolicyCmd = vscode.commands.registerCommand(
```

```
  'safeAppeals.searchPolicy',
  async () =&gt; {
    try {
      const query = await vscode.window.showInputBox({
        prompt: 'Enter your policy question',
        placeHolder: 'e.g., What is the deadline for filing an appeal?'
      });

      if (query) {
        const result = await agentService.executeAgentCall('search_policy', {
          query: query
        });

        if (result.success &amp;&amp; result.answer) {
          // Show result in information message or open in new document
          const action = await vscode.window.showInformationMessage(
            'Policy search completed. View full answer?',
            'View Answer',
            'Dismiss'
          );

          if (action === 'View Answer') {
            const doc = await vscode.workspace.openTextDocument({
              content: `Query: ${query}\n\nAnswer: ${result.answer}\n\nSources:\n${result.results.map((r
              language: 'markdown'
            });
            await vscode.window.showTextDocument(doc);
          }
        } else {
          vscode.window.showWarningMessage('No policy information found for your query.');
        }
      }
    } catch (error) {
      vscode.window.showErrorMessage('Policy search failed: ' + error);
    }
  }
);

// Search workspace command
const searchWorkspaceCmd = vscode.commands.registerCommand(
  'safeAppeals.searchWorkspace',
  async () =&gt; {
    try {
      const query = await vscode.window.showInputBox({
        prompt: 'Search workspace documents',
        placeHolder: 'Enter search terms...'
      });

      if (query) {
        const result = await agentService.executeAgentCall('search_workspace', {
          query: query
        });

        if (result.success &amp;&amp; result.answer) {
          const doc = await vscode.workspace.openTextDocument({
            content: `Workspace Search Results\n\nQuery: ${query}\n\nSummary: ${result.answer}\n\nRelevar
            language: 'markdown'
          });
          await vscode.window.showTextDocument(doc);
        } else {
          vscode.window.showWarningMessage('No relevant documents found in workspace.');
        }
      }
    } catch (error) {
      vscode.window.showErrorMessage('Workspace search failed: ' + error);
    }
  }
);

// Register all commands
context.subscriptions.push(
  uploadPolicyCmd,
```

```
    indexWorkspaceCmd,
    organizeWorkspaceCmd,
    searchPolicyCmd,
    searchWorkspaceCmd
  );
}

export function deactivate() {
  const logger = Logger.getInstance();
  logger.info('Deactivating Safe Appeals Navigator extension');
}
```

Configuration and Templates

Folder Structure Template

resources/templates/workers_compensation.json

```
{
  "Medical Records": {
    "description": "All medical documentation related to the injury",
    "patterns": ["medical", "doctor", "physician", "diagnosis", "treatment", "hospital", "clinic", "mri", "
    "fileTypes": [".pdf", ".docx", ".txt"],
    "subfolders": {
      "Initial Reports": ["initial", "first", "emergency"],
      "Treatment Records": ["treatment", "therapy", "follow"],
      "Diagnostic Reports": ["mri", "ct", "xray", "ultrasound", "diagnostic"]
    }
  },
  "Legal Documents": {
    "description": "Legal filings, court documents, and attorney correspondence",
    "patterns": ["legal", "court", "judgment", "appeal", "hearing", "attorney", "lawyer", "filing", "motion
    "fileTypes": [".pdf", ".docx"],
    "subfolders": {
      "Court Filings": ["filing", "motion", "petition"],
      "Attorney Correspondence": ["attorney", "lawyer", "legal advice"],
      "Judgments and Orders": ["judgment", "order", "decision"]
    }
  },
  "Correspondence": {
    "description": "Email, letters, and other communications",
    "patterns": ["letter", "email", "correspondence", "communication", "reply", "notice"],
    "fileTypes": [".eml", ".txt", ".pdf", ".docx"],
    "subfolders": {
      "Insurance Communications": ["insurance", "adjuster", "claim"],
      "Employer Communications": ["employer", "hr", "workplace"],
      "Government Communications": ["wcb", "government", "ministry"]
    }
  },
  "Policy Documents": {
    "description": "Policy manuals, regulations, and guidelines",
    "patterns": ["policy", "regulation", "rule", "guideline", "procedure", "manual", "act"],
    "fileTypes": [".pdf", ".docx", ".txt"],
    "subfolders": {
      "WCB Policies": ["wcb", "workers compensation"],
      "Employment Policies": ["employment", "workplace", "hr"],
      "Government Regulations": ["regulation", "act", "statute"]
    }
  },
  "Financial Records": {
    "description": "Payment records, invoices, and financial documentation",
    "patterns": ["payment", "invoice", "receipt", "financial", "cost", "bill", "wage", "benefit"],
    "fileTypes": [".pdf", ".xlsx", ".csv", ".txt"],
    "subfolders": {
      "Wage Records": ["wage", "salary", "pay", "earnings"],
      "Medical Expenses": ["medical bill", "invoice", "cost"],
      "Benefit Payments": ["benefit", "compensation", "payment"]
    }
  },
```

```
  "Evidence": {
    "description": "Photos, videos, and other evidence related to the case",
    "patterns": ["photo", "image", "video", "evidence", "witness", "statement"],
    "fileTypes": [".jpg", ".png", ".pdf", ".mp4", ".docx"],
    "subfolders": {
      "Injury Photos": ["injury", "photo", "image"],
      "Workplace Evidence": ["workplace", "scene", "equipment"],
      "Witness Statements": ["witness", "statement", "testimony"]
    }
  },
  "Forms and Applications": {
    "description": "Completed forms and applications",
    "patterns": ["form", "application", "claim", "report", "form", "wcb"],
    "fileTypes": [".pdf", ".docx"],
    "subfolders": {
      "Initial Claim Forms": ["claim", "initial", "first report"],
      "Appeal Forms": ["appeal", "review", "reconsideration"],
      "Medical Forms": ["medical form", "assessment", "evaluation"]
    }
  },
  "Miscellaneous": {
    "description": "Other documents that don't fit specific categories",
    "patterns": ["misc", "other", "general"],
    "fileTypes": [".*"]
  }
}
```

Extension Configuration

package.json (complete configuration section)

```
{
  "contributes": {
    "configuration": {
      "title": "Safe Appeals Navigator",
      "properties": {
        "safeAppeals.openaiApiKey": {
          "type": "string",
          "default": "",
          "description": "OpenAI API key for AI features",
          "scope": "machine"
        },
        "safeAppeals.chromaUrl": {
          "type": "string",
          "default": "http://localhost:8000",
          "description": "Chroma database URL"
        },
        "safeAppeals.chunkSize": {
          "type": "number",
          "default": 500,
          "description": "Text chunk size for document indexing"
        },
        "safeAppeals.chunkOverlap": {
          "type": "number",
          "default": 50,
          "description": "Overlap between text chunks"
        },
        "safeAppeals.searchResultLimit": {
          "type": "number",
          "default": 5,
          "description": "Default number of search results to return"
        },
        "safeAppeals.autoOrganize": {
          "type": "boolean",
          "default": false,
          "description": "Automatically organize files when indexing"
        },
        "safeAppeals.logLevel": {
          "type": "string",
          "enum": ["error", "warn", "info", "debug"],
```

```
        "default": "info",
        "description": "Logging level"
      }
    }
  },
  "commands": [
    {
      "command": "safeAppeals.uploadPolicy",
      "title": "Upload Policy Manual",
      "category": "Safe Appeals"
    },
    {
      "command": "safeAppeals.indexWorkspace",
      "title": "Index Workspace Documents",
      "category": "Safe Appeals"
    },
    {
      "command": "safeAppeals.organizeWorkspace",
      "title": "Organize Workspace Files",
      "category": "Safe Appeals"
    },
    {
      "command": "safeAppeals.searchPolicy",
      "title": "Search Policy Manual",
      "category": "Safe Appeals"
    },
    {
      "command": "safeAppeals.searchWorkspace",
      "title": "Search Workspace Documents",
      "category": "Safe Appeals"
    }
  ],
  "views": {
    "explorer": [
      {
        "id": "safeAppeals.policyDashboard",
        "name": "Policy Manual",
        "when": "true"
      }
    ]
  },
  "menus": {
    "explorer/context": [
      {
        "command": "safeAppeals.uploadPolicy",
        "when": "resourceExtname =~ /\\.(pdf|docx|txt|md)$/",
        "group": "safeAppeals"
      },
      {
        "command": "safeAppeals.indexWorkspace",
        "when": "explorerResourceIsFolder",
        "group": "safeAppeals"
      },
      {
        "command": "safeAppeals.organizeWorkspace",
        "when": "explorerResourceIsFolder",
        "group": "safeAppeals"
      }
    ],
    "commandPalette": [
      {
        "command": "safeAppeals.uploadPolicy",
        "when": "true"
      },
      {
        "command": "safeAppeals.indexWorkspace",
        "when": "workspaceFolderCount &gt; 0"
      },
      {
        "command": "safeAppeals.organizeWorkspace",
        "when": "workspaceFolderCount &gt; 0"
      },
```

```
        {
          "command": "safeAppeals.searchPolicy",
          "when": "true"
        },
        {
          "command": "safeAppeals.searchWorkspace",
          "when": "workspaceFolderCount &gt; 0"
        }
      ]
    },
    "keybindings": [
      {
        "command": "safeAppeals.searchPolicy",
        "key": "ctrl+shift+p",
        "mac": "cmd+shift+p",
        "when": "true"
      },
      {
        "command": "safeAppeals.searchWorkspace",
        "key": "ctrl+shift+w",
        "mac": "cmd+shift+w",
        "when": "workspaceFolderCount &gt; 0"
      }
    ]
  }
}
```

Utility Services

Logging Service

src/utils/logger.ts

```
import * as vscode from 'vscode';
import * as winston from 'winston';
import * as path from 'path';
import { PathResolver } from './pathResolver';

export class Logger {
  private static _instance: Logger;
  private winston: winston.Logger;
  private outputChannel: vscode.OutputChannel;

  private constructor() {
    this.outputChannel = vscode.window.createOutputChannel('Safe Appeals Navigator');

    const pathResolver = PathResolver.getInstance();
    const logPath = path.join(pathResolver.storageDir, 'logs', 'app.log');

    this.winston = winston.createLogger({
      level: vscode.workspace.getConfiguration('safeAppeals').get('logLevel', 'info'),
      format: winston.format.combine(
        winston.format.timestamp(),
        winston.format.errors({ stack: true }),
        winston.format.json()
      ),
      transports: [
        new winston.transports.File({
          filename: logPath,
          maxsize: 10 * 1024 * 1024, // 10MB
          maxFiles: 5
        }),
        new winston.transports.Console({
          format: winston.format.simple()
        })
      ]
    });
  }
```

```
  public static getInstance(): Logger {
    if (!Logger._instance) {
      Logger._instance = new Logger();
    }
    return Logger._instance;
  }

  public info(message: string, ...args: any[]): void {
    const fullMessage = this.formatMessage(message, args);
    this.winston.info(fullMessage);
    this.outputChannel.appendLine(`[INFO] ${fullMessage}`);
  }

  public warn(message: string, ...args: any[]): void {
    const fullMessage = this.formatMessage(message, args);
    this.winston.warn(fullMessage);
    this.outputChannel.appendLine(`[WARN] ${fullMessage}`);
  }

  public error(message: string, error?: any, ...args: any[]): void {
    const fullMessage = this.formatMessage(message, args);
    const errorDetails = error ? ` - ${error.stack || error.message || error}` : '';
    const logMessage = fullMessage + errorDetails;

    this.winston.error(logMessage);
    this.outputChannel.appendLine(`[ERROR] ${logMessage}`);
  }

  public debug(message: string, ...args: any[]): void {
    const fullMessage = this.formatMessage(message, args);
    this.winston.debug(fullMessage);
    this.outputChannel.appendLine(`[DEBUG] ${fullMessage}`);
  }

  private formatMessage(message: string, args: any[]): string {
    if (args.length === 0) return message;

    try {
      return message + ' ' + args.map(arg =&gt;
        typeof arg === 'object' ? JSON.stringify(arg, null, 2) : String(arg)
      ).join(' ');
    } catch {
      return message + ' [formatting error]';
    }
  }

  public show(): void {
    this.outputChannel.show();
  }
}
```

Error Handler

src/utils/errorHandler.ts

```
import * as vscode from 'vscode';
import { Logger } from './logger';

export class ErrorHandler {
  private static logger = Logger.getInstance();

  public static async handle(error: any, context?: string): Promise&lt;void&gt; {
    const contextMsg = context ? `[${context}] ` : '';
    const errorMessage = error instanceof Error ? error.message : String(error);

    this.logger.error(`${contextMsg}${errorMessage}`, error);

    // Show user-friendly error message
    const userMessage = this.getUserFriendlyMessage(error);
```

```
    const action = await vscode.window.showErrorMessage(
      `${contextMsg}${userMessage}`,
      'Show Logs',
      'Dismiss'
    );

    if (action === 'Show Logs') {
      this.logger.show();
    }
  }

  private static getUserFriendlyMessage(error: any): string {
    if (error instanceof Error) {
      if (error.message.includes('OPENAI_API_KEY')) {
        return 'OpenAI API key is missing or invalid. Please configure your API key in settings.';
      }
      if (error.message.includes('ENOENT')) {
        return 'File or directory not found. Please check the file path.';
      }
      if (error.message.includes('EACCES')) {
        return 'Permission denied. Please check file permissions.';
      }
      if (error.message.includes('network')) {
        return 'Network error. Please check your internet connection.';
      }
    }

    return 'An unexpected error occurred. Please check the logs for more details.';
  }

  public static wrap&lt;T&gt;(promise: Promise&lt;T&gt;, context?: string): Promise&lt;T&gt; {
    return promise.catch(error =&gt; {
      this.handle(error, context);
      throw error;
    });
  }
}
```

Testing

Test Configuration

test/runTest.ts

```
import * as path from 'path';
import { runTests } from '@vscode/test-electron';

async function main() {
  try {
    const extensionDevelopmentPath = path.resolve(__dirname, '../../');
    const extensionTestsPath = path.resolve(__dirname, './suite/index');

    await runTests({
      extensionDevelopmentPath,
      extensionTestsPath,
      launchArgs: ['--disable-extensions']
    });
  } catch (err) {
    console.error('Failed to run tests');
    process.exit(1);
  }
}

main();
```

Service Tests

test/suite/indexService.test.ts

```
import * as assert from 'assert';
import * as vscode from 'vscode';
import * as path from 'path';
import * as fs from 'fs/promises';
import { IndexService } from '../../src/services/indexService';

suite('IndexService Test Suite', () =&gt; {
  let indexService: IndexService;

  setup(async () =&gt; {
    indexService = new IndexService();
  });

  teardown(async () =&gt; {
    await indexService.cleanup();
  });

  test('Should index text document', async () =&gt; {
    // Create a test file
    const testContent = 'This is a test document for indexing.';
    const testFilePath = path.join(__dirname, 'test.txt');

    await fs.writeFile(testFilePath, testContent);

    try {
      const docId = await indexService.indexDocument(testFilePath, false);
      assert.ok(docId);
      assert.ok(typeof docId === 'string');
    } finally {
      await fs.unlink(testFilePath);
    }
  });

  test('Should search indexed documents', async () =&gt; {
    const testContent = 'Workers compensation policy regarding medical benefits.';
    const testFilePath = path.join(__dirname, 'policy.txt');

    await fs.writeFile(testFilePath, testContent);

    try {
      await indexService.indexDocument(testFilePath, true);
      const results = await indexService.searchSimilar('medical benefits', 'policy_manual', 5);

      assert.ok(Array.isArray(results));
    } finally {
      await fs.unlink(testFilePath);
    }
  });
});
```

Build and Deployment

Build Scripts

scripts/build.js

```
const { build } = require('esbuild');
const { dependencies } = require('../package.json');

const shared = {
  bundle: true,
  entryPoints: ['src/extension.ts'],
  external: ['vscode'],
  format: 'cjs',
```

```
  platform: 'node',
  target: 'node16'
};

// Development build
build({
  ...shared,
  outfile: 'out/extension.js',
  sourcemap: true
}).catch(() =&gt; process.exit(1));

// Production build
if (process.argv.includes('--production')) {
  build({
    ...shared,
    outfile: 'out/extension.js',
    minify: true,
    sourcemap: false,
    define: {
      'process.env.NODE_ENV': '"production"'
    }
  }).catch(() =&gt; process.exit(1));
}
```

Package Configuration

.vscodeignore

```
.vscode/**
.vscode-test/**
src/**
test/**
node_modules/**
*.map
.gitignore
.eslintrc.json
tsconfig.json
webpack.config.js
scripts/**
resources/templates/**
*.log
```

Environment Setup

Environment Variables

Create `.env.example`:

```
# OpenAI API Configuration<a></a>
OPENAI_API_KEY=your_openai_api_key_here

# Chroma Configuration<a></a>
CHROMA_HOST=localhost
CHROMA_PORT=8000
CHROMA_SSL=false

# Application Configuration<a></a>
LOG_LEVEL=info
STORAGE_PATH=

# Development Configuration<a></a>
NODE_ENV=development
DEBUG_MODE=false
```

Launch Configuration

.vscode/launch.json

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Run Extension",
      "type": "extensionHost",
      "request": "launch",
      "args": ["--extensionDevelopmentPath=${workspaceFolder}"],
      "outFiles": ["${workspaceFolder}/out/**/*.js"],
      "preLaunchTask": "${workspaceFolder}/npm: watch"
    },
    {
      "name": "Extension Tests",
      "type": "extensionHost",
      "request": "launch",
      "args": [
        "--extensionDevelopmentPath=${workspaceFolder}",
        "--extensionTestsPath=${workspaceFolder}/out/test/suite/index"
      ],
      "outFiles": ["${workspaceFolder}/out/test/**/*.js"],
      "preLaunchTask": "${workspaceFolder}/npm: watch"
    }
  ]
}
```

This comprehensive guide provides all the necessary code, configuration, and implementation details needed to build a fully functional RAG system with Chroma and SQLite integration for Safe Appeals Navigator. The system includes document indexing, AI-powered search, automatic file organization, and a complete policy management dashboard.