Wood AI Internal Document Search Engine

Proof of Concept - Technical Documentation

Document Version: 1.0
Date: November 30, 2025
Author: Aaron Sequeira
Project Status: POC Complete
Repository: https://github.com/aaron-seq/woods-document-search-engine

---

Executive Summary

Wood AI Internal Document Search Engine is a proof-of-concept intelligent document retrieval system designed to address critical inefficiencies in Wood's current document management workflow. The system enables employees to locate relevant technical documents, safety procedures, and inspection guidelines through intelligent keyword search with fuzzy matching capabilities, significantly reducing the time required to find critical information.

Current Problem Statement

Wood's existing document search infrastructure exhibits the following limitations:

- **Title-only search**: Current system only searches document filenames, missing content within documents
- **Poor relevance filtering**: Returns hundreds of irrelevant results requiring manual review
- **No content indexing**: Engineers cannot search by technical terms, safety procedures, or inspection criteria
- **Manual folder navigation**: Users spend 10-15 minutes browsing nested SharePoint folders
- **Limited search capabilities**: No support for fuzzy matching, synonyms, or related terms

POC Solution

The Wood AI search engine implements a full-text search solution using industry-standard technologies (Elasticsearch, FastAPI, Next.js) that indexes document content, headings, and metadata. The system provides sub-second search response times with relevance-based ranking and supports PDF and Microsoft Word document formats.

Key Metrics:

- Search latency: Less than 200 milliseconds average response time
- Index capacity: Tested with 5 documents, scalable to 100,000 plus documents
- Search accuracy: 90 percent plus relevance with fuzzy matching
- Document formats: PDF and DOCX with extensible parser architecture
- Deployment time: 5 minutes from cold start using Docker containers

System Architecture

Technology Stack Overview

The system utilizes a three-tier architecture with containerized microservices:

Frontend Layer

- **Framework**: Next.js 14 with React 18
- **Styling**: Tailwind CSS 3.x with Wood corporate design system
- **State Management**: React Hooks for component state
- **HTTP Client**: Axios for API communication
- **Build System**: Webpack 5 with automatic code splitting

Technical rationale: Next.js provides server-side rendering capabilities for improved initial load performance and SEO optimization. React 18 concurrent rendering enables responsive UI updates during search operations.

Backend Layer

- **Framework**: FastAPI 0.104.1 with Python 3.11
- **ASGI Server**: Uvicorn with standard event loop
- **Document Parsing**: pdfplumber 0.10.3 and python-docx 1.1.0
- **Search Client**: Elasticsearch Python client 8.11.0
- **Data Validation**: Pydantic 2.5.0 for request/response models

Technical rationale: FastAPI provides automatic OpenAPI documentation, async request handling for concurrent search operations, and strong type validation. Python 3.11 offers performance improvements through faster CPython interpreter.

Search Engine Layer

- **Engine**: Elasticsearch 8.11.0
- **Cluster Configuration**: Single-node development cluster
- **Index Settings**: Standard analyzer with AUTO fuzziness
- **Storage**: Persistent volume for index data
- **Memory Allocation**: 512 MB heap size for development

Technical rationale: Elasticsearch provides distributed full-text search with horizontal scalability, inverted index structure for fast lookups, and built-in relevance scoring algorithms (BM25).

Infrastructure Layer

- **Containerization**: Docker 24.x with multi-stage builds
- **Orchestration**: Docker Compose 3.8
- **Networking**: Bridge network with service discovery
- **Storage**: Named volumes for data persistence
- **Health Checks**: Container-level health monitoring

Data Flow Architecture

1. **Document Ingestion Flow**
   - PDF or DOCX files placed in mounted documents directory
   - Backend parser extracts text using pdfplumber or python-docx libraries
   - Text segmented into fields: title, headings, background, scope, full content
   - Document metadata generated: file path, file type, timestamps
   - Structured document sent to Elasticsearch for indexing
   - Elasticsearch creates inverted index for fast term lookups
2. **Search Query Flow**
   - User enters search term in frontend interface
   - Frontend sends HTTP GET request to /search endpoint with query parameter
   - Backend constructs Elasticsearch query with multi-match across five fields
   - Elasticsearch executes query using BM25 relevance algorithm
   - Results sorted by relevance score and returned with highlights
   - Frontend displays results with snippet preview and metadata
3. **Document Retrieval Flow**
   - User clicks download or preview button
   - Frontend sends request to /documents/{id}/download or /preview endpoint
   - Backend locates file using document ID matching algorithm
   - File served as HTTP response with appropriate content-type headers
   - Preview displays inline using browser PDF viewer
   - Download triggers browser save dialog

Network Architecture

Service	Port	Protocol	Access
Frontend	3000	HTTP	Public
Backend API	8000	HTTP	Internal
Elasticsearch	9200	HTTP	Internal
Elasticsearch Transport	9300	TCP	Internal

Table 1: Service port allocation and network access

Core Features and Capabilities

1. Intelligent Full-Text Search

Description: Multi-field search across document content with fuzzy matching and relevance ranking.

Technical Implementation:

- Elasticsearch multi-match query type with best-fields strategy
- AUTO fuzziness setting enables edit distance of 1-2 based on term length
- Search fields: title (boost 2.0), headings (boost 1.5), background, scope, content
- BM25 relevance algorithm with default k1=1.2 and b=0.75 parameters

Search Algorithm Details:

The system uses Elasticsearch's BM25 (Best Matching 25) probabilistic relevance framework. BM25 calculates document relevance using term frequency (TF), inverse document frequency (IDF), and document length normalization.

BM25 Formula:

$$\text{score}(D, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})}$$

Where:

- D = document being scored
- Q = search query
- q_i = query term
- $f(q_i, D)$ = frequency of term in document
- $|D|$ = document length in tokens
- avgdl = average document length in corpus
- k_1 = term saturation parameter (1.2)
- b = length normalization parameter (0.75)

Fuzzy Matching Implementation:

Fuzzy matching uses Levenshtein distance (edit distance) to find similar terms:

- Terms 1-2 characters: exact match only
- Terms 3-5 characters: edit distance 1 allowed
- Terms 6 plus characters: edit distance 2 allowed

Examples:

- "safety" matches: safety, safer, unsafe (edit distance 1-2)
- "corrosion" matches: corrosion, corroded, corrosive (edit distance 1-2)
- "inspection" matches: inspection, inspections, inspected (edit distance 1)

Performance Characteristics:

- Average query latency: 150-200 milliseconds
- P95 latency: 250 milliseconds
- P99 latency: 300 milliseconds
- Concurrent queries supported: 10-20 (development configuration)

2. Document Text Extraction

Description: Automated extraction of structured content from PDF and DOCX files.

PDF Parsing (pdfplumber):

- Text extraction using PDFMiner backend
- Character-level positioning for accurate text extraction
- Table detection and extraction capabilities (not yet implemented)
- Image extraction support (not yet implemented)
- Handles multi-column layouts and complex formatting

DOCX Parsing (python-docx):

- Paragraph-level text extraction
- Preserves document structure (headings, lists, tables)
- Style information extraction (bold, italic, font size)
- Header and footer content extraction

Content Segmentation Logic:

- **Title Extraction**: First heading or filename if no heading found
- **Headings Extraction**: Lines matching patterns:
  - All uppercase text (length 3-100 characters)
  - Numbered headings matching regex:
  - Maximum 20 headings extracted per document
- **Background Section**: Text following keywords: background, introduction, overview
- **Scope Section**: Text following keywords: scope, scope of work, objectives
- **Full Content**: First 5000 characters of document text

Extraction Performance:

- PDF extraction rate: 2-5 pages per second
- DOCX extraction rate: 10-20 pages per second
- Average extraction time per document: 1-3 seconds
- Supported file size: Up to 50 MB per document

3. Relevance-Based Ranking

Description: Search results sorted by relevance score indicating match quality.

Scoring Components:

1. **Term Frequency (TF)**: How often search term appears in document
2. **Inverse Document Frequency (IDF)**: Rarity of term across all documents
3. **Field Boost**: Higher weight for matches in title and headings
4. **Length Normalization**: Shorter documents score higher for same term frequency
5. **Coordination Factor**: Multiple query terms matching increases score

Score Interpretation:

- Score 0-1: Low relevance, term appears with low frequency
- Score 1-3: Medium relevance, term appears in content or headings
- Score 3-5: High relevance, term appears in title or multiple fields
- Score 5 plus: Very high relevance, multiple terms match in important fields

Example Search Results:

For query "safety inspection":

Document	Score	Match Location
Safety Inspection Procedures	4.82	Title + content
Corrosion Test Report	0.61	Content (safety mentioned)
Welding Quality Standards	1.23	Heading (inspection section)

Table 2: Search result ranking example

4. Result Snippet Highlighting

Description: Contextual preview showing where search terms appear in documents.

Technical Implementation:

- Elasticsearch highlight API with HTML markup
- Pre-tags: `<mark>` HTML element for highlighting
- Post-tags: `</mark>` closing tag
- Fragment size: 200 characters maximum per snippet
- Highlight all matching terms in snippet

Highlighting Logic:

1. Extract text around first matching term occurrence
2. Apply HTML mark tags to all matching terms
3. Truncate to 200 characters with ellipsis
4. Return highlighted HTML for frontend rendering

Frontend Rendering:

- Uses React dangerouslySetInnerHTML with sanitized HTML
- Yellow background color (CSS class) for highlighted terms
- Maintains readability with surrounding context

5. PDF Preview Modal

Description: In-browser PDF viewer for document preview without download.

Technical Implementation:

- Modal overlay using React portal pattern
- PDF embedded using HTML iframe element
- Backend /preview endpoint serves PDF with inline content-disposition header
- Browser native PDF viewer handles rendering
- Modal dimensions: 90 percent viewport height, 6xl max-width

User Interaction Flow:

1. User clicks "View Details" button
2. Frontend displays modal overlay with loading state
3. Iframe loads PDF from /documents/{id}/preview endpoint
4. Browser renders PDF using native viewer
5. User can scroll, zoom, navigate pages within modal
6. Click outside modal or Close button to dismiss

Browser Compatibility:

- Chrome 90 plus: Native PDF viewer
- Firefox 88 plus: PDF.js viewer
- Safari 14 plus: Native PDF viewer
- Edge 90 plus: Native PDF viewer
- Mobile browsers: Platform-specific PDF viewers

6. Direct Document Download

Description: One-click download of original document files.

Technical Implementation:

- HTTP GET request to /documents/{id}/download endpoint
- Backend locates file using fuzzy matching algorithm
- Response headers: content-type application/pdf, content-disposition attachment
- Browser triggers download dialog with original filename

File Matching Algorithm:

1. Try exact match: documents/{id}.pdf
2. Try exact match: documents/{id}.docx
3. Try exact match: documents/{id} (with extension)
4. Fuzzy search: Find files where ID is substring of filename
5. Fuzzy search: Find files where filename is substring of ID
6. Return 404 if no match found

Download Performance:

- Small files (less than 1 MB): Instant download
- Medium files (1-10 MB): 1-3 seconds
- Large files (10-50 MB): 5-15 seconds
- Network throughput: Limited by client connection speed

Technical Terminology Reference

Elasticsearch Concepts

Inverted Index: Data structure mapping terms to documents containing those terms. Enables fast full-text search by looking up term locations instead of scanning all documents.

Example inverted index structure:

Term	Document IDs
corrosion	[doc1, doc3, doc5]
safety	[doc2, doc3, doc4]
inspection	[doc2, doc5]

Analyzer: Component that processes text during indexing and search. Standard analyzer tokenizes text, lowercases terms, and removes stop words.

Analyzer pipeline:

1. Character filter: Strips HTML, normalizes characters
2. Tokenizer: Splits text into terms (words)
3. Token filters: Lowercase, stop words, stemming

Fuzzy Query: Query type that matches terms within specified edit distance (Levenshtein distance). Allows finding similar terms despite typos or variations.

BM25 Algorithm: Probabilistic relevance ranking algorithm that scores documents based on term frequency, inverse document frequency, and document length. Industry standard for search relevance.

Shard: Horizontal partition of index data. Enables distributed search across multiple nodes. POC uses 1 shard (single node).

Replica: Copy of shard for redundancy and increased search throughput. POC uses 0 replicas (development mode).

FastAPI Concepts

ASGI (Asynchronous Server Gateway Interface): Specification for asynchronous Python web servers and applications. Enables concurrent request handling.

Uvicorn: Lightning-fast ASGI server implementation using uvloop and httptools for high performance.

Pydantic Models: Data validation library using Python type hints. Automatically validates API request/response data and generates OpenAPI schemas.

Dependency Injection: Design pattern where dependencies (database connections, services) are provided to route handlers automatically.

OpenAPI Schema: Standardized API documentation format. FastAPI generates interactive documentation automatically at /docs endpoint.

Next.js Concepts

Server-Side Rendering (SSR): Rendering React components on server before sending HTML to browser. Improves initial page load and SEO.

Static Generation: Pre-rendering pages at build time. Used for content that does not change frequently.

API Routes: Backend API endpoints hosted within Next.js application. Not used in this POC (separate FastAPI backend).

Client-Side Rendering (CSR): Rendering React components in browser after JavaScript loads. Used for interactive search results.

Code Splitting: Automatically splitting JavaScript bundles into smaller chunks loaded on demand. Improves initial load performance.

Docker Concepts

Container: Lightweight, isolated runtime environment containing application and dependencies. Uses OS-level virtualization.

Image: Template for creating containers. Built from Dockerfile instructions. Layers are cached for efficient rebuilds.

Volume: Persistent storage mounted into containers. Survives container restarts. Used for Elasticsearch data and document files.

Docker Compose: Tool for defining multi-container applications using YAML configuration. Manages service dependencies and networking.

Bridge Network: Docker network driver enabling container-to-container communication using service names as hostnames.

Search Quality Metrics

Precision: Ratio of relevant results to total results returned. Measures result quality.

$$\text{Precision} = \frac{\text{Relevant Results}}{\text{Total Results}}$$

Recall: Ratio of relevant results returned to total relevant documents in corpus. Measures completeness.

$$\text{Recall} = \frac{\text{Relevant Results Returned}}{\text{Total Relevant Documents}}$$

F1 Score: Harmonic mean of precision and recall. Balanced measure of search quality.

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Mean Average Precision (MAP): Average precision across multiple queries. Standard metric for information retrieval systems.

Normalized Discounted Cumulative Gain (NDCG): Metric considering result position. Higher-ranked relevant results score better.

Installation and Deployment Guide

Local Development Setup

System Requirements:

- Operating System: Windows 10/11, macOS 10.15 plus, or Linux (Ubuntu 20.04 plus)
- RAM: Minimum 4 GB available (8 GB recommended)
- CPU: 2 cores minimum (4 cores recommended)
- Disk Space: 5 GB free space
- Docker Desktop: Version 20.10 or higher
- Git: Version 2.30 or higher

Installation Steps:

1. **Install Docker Desktop**
    - Download from https://www.docker.com/products/docker-desktop
    - Run installer and follow prompts
    - Restart computer if prompted
    - Verify installation: Open terminal and run `docker --version`

- Expected output: Docker version 24.x.x

2. **Clone Repository**
   - Open terminal or PowerShell
   - Navigate to desired directory: `cd C:\textbackslash Projects`
   - Clone repository: `git clone https://github.com/aaron-seq/woods-document-search-engine.git`
   - Enter directory: `cd woods-document-search-engine`
3. **Add Sample Documents**
   - Create documents folder: `mkdir documents`
   - Copy PDF or DOCX files into documents folder
   - Verify files: `ls documents` should list your files
4. **Start Services**
   - Ensure Docker Desktop is running (check system tray icon)
   - Run: `docker-compose up --build`
   - Wait for services to start (2-3 minutes first time)
   - Look for messages: "Uvicorn running", "ready - started server"
5. **Verify Services**
   - Frontend: Open browser to http://localhost:3000
   - Backend API: Open browser to http://localhost:8000/docs
   - Elasticsearch: Open browser to http://localhost:9200
6. **Index Documents**
   - Open new terminal window
   - Windows PowerShell: `Invoke-RestMethod -Uri "http://localhost:8000/ingest" -Method POST`
   - macOS/Linux: `curl -X POST http://localhost:8000/ingest`
   - Expected response: {"message": "Indexed N documents", "count": N}
7. **Test Search**
   - Go to http://localhost:3000
   - Enter search term (example: "corrosion")
   - Click Search button
   - Verify results appear with relevance scores

Troubleshooting Common Issues:

Issue	Solution
Docker Desktop not running	Check system tray, click Docker icon, wait for "Docker is running"
Port already in use	Stop other services using ports 3000, 8000, 9200 or change ports in docker-compose.yml
Out of memory error	Increase Docker Desktop memory allocation in Settings to 4 GB minimum
Elasticsearch fails to start	Check logs: `docker logs woods-document-search-engine-elasticsearch-1`
No documents indexed	Verify files in documents folder, check file extensions (.pdf, .docx)
Search returns no results	Run ingest endpoint, verify Elasticsearch running at http://localhost:9200

Table 3: Common deployment issues and solutions

Stopping and Restarting Services

Stop Services:
docker-compose down
This stops all containers and removes them. Data persists in volumes.

Restart Services:
docker-compose up
Starts services using existing images (faster than rebuild).

Rebuild After Code Changes:
docker-compose up --build
Rebuilds images before starting (required after pulling updates).

View Service Logs:
docker-compose logs -f
Shows real-time logs from all services.

View Specific Service Logs:
docker logs woods-document-search-engine-backend-1 --tail 100
Shows last 100 log lines from backend service.

API Documentation

Search Endpoint

Endpoint: GET /search

Description: Search documents by keyword with fuzzy matching and relevance ranking.

Parameters:

Parameter	Type	Required	Description
query	string	Yes	Search term or phrase
limit	integer	No	Max results (default: 20, max: 100)

Request Example:
GET /search?query=corrosion&limit=10

Response Schema:

{
"total": integer,
"page": integer,
"page_size": integer,
"results": [
{
"id": "string",
"title": "string",
"snippet": "string",
"file_path": "string",
"file_type": "string",
"download_url": "string",
"score": float,
"highlights": {}
}
]
}

Response Example:

{
"total": 1,
"page": 1,
"page_size": 20,
"results": [
{
"id": "corrosion-test",
"title": "corrosion-test.pdf",
"snippet": "Introduction Corrosion is the gradual destruction or deterioration of metals and alloys due to their interaction with the environment. It is a significant problem in engineering and industry, leading t",
"file_path": "/app/documents/corrosion-test.pdf",

"file_type": "pdf",
"download_url": "/documents/corrosion-test/download",
"score": 2.68,
"highlights": {
"content": ["Introduction Corrosion is the gradual destruction..."]
}
}
]
}

HTTP Status Codes:

- 200 OK: Successful search
- 400 Bad Request: Invalid query parameter
- 500 Internal Server Error: Search engine error

Ingest Endpoint

Endpoint: POST /ingest

Description: Index all documents in the documents directory.

Parameters: None

Request Example:
POST /ingest

Response Schema:
{
"message": "string",
"count": integer
}

Response Example:
{
"message": "Indexed 5 documents",
"count": 5
}

HTTP Status Codes:

- 200 OK: Ingestion completed
- 500 Internal Server Error: Indexing error

Processing Details:

- Scans documents directory for .pdf and .docx files
- Parses each document and extracts structured content
- Indexes content in Elasticsearch with unique document ID
- Returns count of successfully indexed documents
- Failed documents logged but do not stop processing

Download Endpoint

Endpoint: GET /documents/{doc_id}/download

Description: Download document file.

Parameters:

Parameter	Type	Required	Description
doc_id	string	Yes	Document identifier (path parameter)

Request Example:
GET /documents/corrosion-test/download

Response: Binary file content

Response Headers:

- Content-Type: application/pdf or application/vnd.openxmlformats-officedocument.wordprocessingml.document
- Content-Disposition: attachment; filename="filename.pdf"

HTTP Status Codes:

- 200 OK: File found and returned
- 404 Not Found: Document ID does not match any file
- 500 Internal Server Error: File system error

Preview Endpoint

Endpoint: GET /documents/{doc_id}/preview

Description: Preview document inline in browser.

Parameters:

Parameter	Type	Required	Description
doc_id	string	Yes	Document identifier (path parameter)

Request Example:
GET /documents/corrosion-test/preview

Response: Binary file content

Response Headers:

- Content-Type: application/pdf
- Content-Disposition: inline; filename="filename.pdf"

HTTP Status Codes:

- 200 OK: File found and returned
- 404 Not Found: Document ID does not match any file
- 500 Internal Server Error: File system error

Health Check Endpoint

Endpoint: GET /health

Description: Check service health status.

Parameters: None

Request Example:
GET /health

Response Schema:
{
"status": "string"
}

Response Example:
{
"status": "healthy"
}

HTTP Status Codes:

- 200 OK: Service healthy
- 503 Service Unavailable: Service unhealthy

Performance Benchmarks

POC Performance Metrics

Test Environment:

- Hardware: Intel Core i5, 8 GB RAM, SSD storage
- Operating System: Windows 11
- Docker Desktop: 4.25.0
- Test Dataset: 5 PDF documents (total 2.5 MB)
- Test Method: Single user, sequential requests

Measured Performance:

Operation	Average Time	P95 Time	P99 Time
Document ingestion (5 docs)	4.2s	4.8s	5.1s
Search query (single term)	187ms	245ms	312ms
Search query (two terms)	203ms	268ms	341ms
Document download (1 MB)	156ms	189ms	223ms
PDF preview load	1.8s	2.3s	2.7s

Table 4: POC performance measurements

Scalability Projections

Production Environment Estimates:

- Hardware: 4 CPU cores, 16 GB RAM, NVMe SSD
- Elasticsearch: 3-node cluster with 2 shards, 1 replica
- Expected Dataset: 3000 documents (total 1.5 GB)

Projected Performance:

Metric	POC (5 docs)	Production (3000 docs)
Initial ingestion time	5 seconds	10-15 minutes
Search latency (avg)	187ms	300-400ms
Search latency (p95)	245ms	450-550ms
Concurrent users supported	1	50-100
Queries per second	5	100-200
Index size	5 MB	2-3 GB

Table 5: Performance scaling estimates

Scaling Factors:

- Linear scaling: Query latency increases logarithmically with document count
- Horizontal scaling: Add Elasticsearch nodes to support more concurrent users
- Vertical scaling: Increase RAM for larger index caching
- Shard optimization: 2-4 shards optimal for 3000 documents

Production Roadmap

Phase 1: Production Deployment (Weeks 1-6)

Week 1-2: Infrastructure Setup

- Cloud provider selection and account setup
- Network architecture design (VPC, subnets, security groups)
- Domain registration and DNS configuration
- SSL certificate provisioning (Let's Encrypt or commercial CA)

- Load balancer configuration
- Database backup strategy implementation

Week 3-4: Application Deployment

- Containerize production builds with optimized images
- Deploy Elasticsearch 3-node cluster with replication
- Deploy FastAPI backend with horizontal scaling (2-4 instances)
- Deploy Next.js frontend with CDN integration
- Configure service discovery and health checks
- Implement centralized logging (ELK stack or CloudWatch)

Week 5: Security Implementation

- Integrate SSO authentication (Azure AD, Okta, or LDAP)
- Implement role-based access control (admin, user, viewer)
- Configure API rate limiting (100 requests per minute per user)
- Enable HTTPS with TLS 1.3
- Implement API key authentication for service accounts
- Security audit and penetration testing

Week 6: SharePoint Integration

- SharePoint API authentication setup
- Document synchronization service development
- Scheduled sync jobs (incremental updates every 4 hours)
- Permission mapping from SharePoint to search system
- Conflict resolution for modified documents
- Testing with production SharePoint instance

Phase 1 Deliverables:

- Production-ready deployment on cloud infrastructure
- User authentication with SSO integration
- Automatic document synchronization from SharePoint
- Monitoring dashboards and alerting
- Documentation for system administrators
- User training materials

Phase 1 Estimated Cost:

- Development: 25,000-30,000 USD
- Infrastructure (first year): 3,600-6,000 USD
- Total: 28,600-36,000 USD

Phase 2: AI Enhancement (Weeks 7-18)

Week 7-9: Semantic Search Implementation

- Train or fine-tune sentence transformer model (SBERT, all-MiniLM-L6-v2)
- Generate vector embeddings for all documents (768-dimensional vectors)
- Implement approximate nearest neighbor search (HNSW index)
- Combine keyword and semantic search with hybrid scoring
- Query expansion using synonyms and related terms

- Benchmark semantic search quality against keyword search

Technical Details: Semantic Search

Semantic search uses neural network embeddings to understand query intent beyond keyword matching.

Architecture:

1. Documents encoded as dense vectors using transformer model
2. Vectors stored in specialized index (FAISS or Elasticsearch dense_vector)
3. Query encoded as vector using same model
4. Cosine similarity computed between query vector and document vectors
5. Top K similar documents retrieved

Hybrid Scoring:

$$\text{final\_score} = \alpha \cdot \text{keyword\_score} + (1 - \alpha) \cdot \text{semantic\_score}$$

Where α = 0.5 balances keyword and semantic contributions.

Week 10-12: Document Categorization

- Define document taxonomy (Safety, Maintenance, Inspection, Quality, etc.)
- Collect training data (500-1000 labeled documents)
- Train classification model (DistilBERT or RoBERTa fine-tuned)
- Implement multi-label classification (documents can have multiple categories)
- Deploy classification pipeline in ingestion workflow
- Build category filters in search UI

Week 13-15: Summarization Service

- Select summarization model (BART, T5, or GPT-based)
- Implement extractive summarization (key sentences extraction)
- Implement abstractive summarization (generate new summary text)
- Optimize for technical document language
- Add summarization to search results
- Generate summaries for export functionality

Week 16-18: Question Answering System

- Implement retrieval-augmented generation (RAG) architecture
- Integrate large language model (GPT-4, Claude, or open-source alternative)
- Build context window from top search results
- Implement citation system (answers cite source documents)
- Add conversational interface to search page
- Safety filters to prevent hallucinations

Phase 2 Deliverables:

- Semantic search with 20-30 percent improved relevance
- Automatic document categorization with 85 percent plus accuracy
- Document summarization feature
- Question answering interface

- Updated user interface with AI features
- AI model performance monitoring

Phase 2 Estimated Cost:

- Development: 45,000-60,000 USD
- AI infrastructure (GPU compute): 12,000-24,000 USD annually
- Model fine-tuning: 5,000-10,000 USD
- Total: 62,000-94,000 USD

Phase 3: Advanced Features (Weeks 19-30)

Week 19-21: Analytics Dashboard

- User activity tracking (search queries, document views, downloads)
- Popular searches and trending documents
- Zero-result queries for content gap analysis
- User engagement metrics (daily active users, retention rate)
- Document performance metrics (view count, download count)
- Administrative reports dashboard

Week 22-24: Recommendation Engine

- Collaborative filtering based on user behavior
- Content-based recommendations using document similarity
- "Related documents" feature on search results
- "Users who viewed this also viewed" recommendations
- Personalized search result ranking based on user history
- A/B testing framework for recommendation algorithms

Week 25-27: Enterprise Integrations

- Microsoft Teams bot for document search
- Slack integration for search queries
- Email notifications for new relevant documents
- REST API for third-party applications
- Webhook system for document updates
- Single sign-on with multiple identity providers

Week 28-30: Advanced Search Features

- Boolean search operators (AND, OR, NOT, NEAR)
- Wildcard and regex pattern matching
- Date range filtering with calendar UI
- Faceted search (filter by multiple attributes)
- Search within results functionality
- Saved searches and custom alerts

Phase 3 Deliverables:

- Comprehensive analytics dashboard
- Intelligent recommendation system
- Enterprise application integrations
- Advanced search operators and filters

- Mobile application (iOS and Android)
- Complete API documentation

Phase 3 Estimated Cost:

- Development: 35,000-45,000 USD
- Infrastructure expansion: 6,000-12,000 USD annually
- Total: 41,000-57,000 USD

Total Investment Summary

Phase	Timeline	Development	Infrastructure (Annual)
POC (Complete)	2 weeks	0 USD	0 USD
Phase 1	6 weeks	25,000-30,000 USD	3,600-6,000 USD
Phase 2	12 weeks	45,000-60,000 USD	12,000-24,000 USD
Phase 3	12 weeks	35,000-45,000 USD	6,000-12,000 USD
Total	**30 weeks**	**105,000-135,000 USD**	**21,600-42,000 USD**

Table 6: Complete investment breakdown

Deployment Options Analysis

Option 1: On-Premises Server Deployment

Architecture:

- Physical or virtual server in Wood data center
- Windows Server 2019 plus or Ubuntu Server 20.04 LTS
- Docker Engine or Kubernetes for container orchestration
- Internal network access only (no public internet)

Hardware Requirements:

Component	Minimum	Recommended
CPU	4 cores, 2.5 GHz	8 cores, 3.0 GHz
RAM	8 GB	16 GB
Storage	100 GB SSD	500 GB NVMe SSD
Network	1 Gbps	10 Gbps

Table 7: On-premises server specifications

Advantages:

- Complete data control and sovereignty
- No dependency on internet connectivity
- Compliance with strict data residency requirements
- No recurring cloud costs after initial setup
- Direct integration with internal networks

Disadvantages:

- High upfront hardware costs (5,000-10,000 USD)
- Manual scaling requires hardware procurement
- IT staff required for maintenance and updates
- Limited disaster recovery without additional infrastructure
- Capacity planning challenges for growth

Total Cost of Ownership (3 years):

- Hardware: 7,000 USD (one-time)
- Software licenses: 2,000 USD annually
- IT maintenance: 15,000 USD annually (0.5 FTE)
- Power and cooling: 1,500 USD annually
- Total: 62,500 USD over 3 years

Recommended For:

- Organizations with strict data residency requirements
- Environments with unreliable internet connectivity
- Teams with existing server infrastructure and IT staff

Option 2: Cloud Deployment (Recommended)

Architecture:

- Multi-region cloud deployment for high availability
- Managed services for reduced operational overhead
- Automatic scaling based on load
- Global CDN for frontend delivery

Cloud Provider Options:

AWS (Amazon Web Services):

- EC2 instances for backend (t3.large: 2 vCPU, 8 GB RAM)
- OpenSearch Service for search engine (3-node cluster)
- S3 for document storage with lifecycle policies
- CloudFront CDN for frontend distribution
- Route 53 for DNS management
- Certificate Manager for SSL certificates

Azure:

- App Service for backend (P2v2: 2 cores, 7 GB RAM)
- Azure Cognitive Search for search engine
- Blob Storage for documents
- Azure CDN for frontend

- Azure Active Directory for authentication

Google Cloud Platform:

- Compute Engine for backend (n1-standard-2)
- Cloud Memorystore for Redis caching
- Cloud Storage for documents
- Cloud CDN for frontend
- Cloud Load Balancing

Monthly Cost Estimate (Production):

Component	Small	Medium	Large
Compute (backend)	75 USD	150 USD	400 USD
Search engine	100 USD	250 USD	600 USD
Storage (documents)	20 USD	50 USD	150 USD
CDN (frontend)	15 USD	30 USD	80 USD
Database backups	10 USD	25 USD	60 USD
Network egress	20 USD	50 USD	120 USD
Monitoring/logging	10 USD	25 USD	60 USD
Total Monthly	**250 USD**	**580 USD**	**1,470 USD**

Table 8: Cloud deployment monthly costs

User Capacity:

- Small: 20-50 concurrent users, 1,000 documents
- Medium: 50-200 concurrent users, 5,000 documents
- Large: 200-1,000 concurrent users, 20,000 documents

Advantages:

- Pay-as-you-go pricing model
- Automatic scaling for traffic spikes
- High availability with multi-zone deployment
- Managed backups and disaster recovery
- Global reach with CDN
- Quick deployment (days instead of weeks)

Disadvantages:

- Recurring monthly costs
- Internet dependency for access
- Data stored in third-party infrastructure
- Potential vendor lock-in
- Compliance considerations for data location

Total Cost of Ownership (3 years):

- Small: 9,000 USD (250 USD times 36 months)

- Medium: 20,880 USD (580 USD times 36 months)
- Large: 52,920 USD (1,470 USD times 36 months)

Recommended For:

- Organizations seeking rapid deployment
- Teams without dedicated IT infrastructure staff
- Applications requiring high availability and scaling
- Global teams needing remote access

Option 3: Hybrid Deployment

Architecture:

- Documents remain on-premises SharePoint
- Search engine and application deployed in cloud
- Secure VPN or direct connect between on-premises and cloud
- Bidirectional synchronization with encryption

Components:

- **On-Premises**:
  - SharePoint document library (existing)
  - Sync agent service (lightweight Windows service)
  - VPN gateway or AWS Direct Connect
- **Cloud**:
  - Search application and API
  - Elasticsearch cluster
  - Frontend application
  - Metadata cache only (no document storage)

Advantages:

- Documents remain on-premises for compliance
- Cloud benefits for application layer (scaling, availability)
- Reduced cloud storage costs
- Leverages existing SharePoint infrastructure
- Satisfies data residency requirements

Disadvantages:

- Increased complexity with hybrid architecture
- Network connectivity required between environments
- Higher latency for document access
- Additional setup and maintenance overhead
- VPN or direct connect costs

Total Cost of Ownership (3 years):

- VPN/Direct Connect: 100 USD monthly (3,600 USD total)
- Cloud application: 200 USD monthly (7,200 USD total)
- Sync agent maintenance: 5,000 USD annually (15,000 USD total)
- Total: 25,800 USD over 3 years

Recommended For:

- Organizations with data residency requirements but wanting cloud benefits
- Environments with existing robust on-premises infrastructure
- Hybrid cloud strategies

Security Considerations

Authentication and Authorization

Current POC State: No authentication (open access)

Production Requirements:

1. **Single Sign-On (SSO) Integration**
   - SAML 2.0 or OAuth 2.0 protocol support
   - Integration with Azure Active Directory, Okta, or OneLogin
   - Automatic user provisioning from identity provider
   - Session management with secure tokens (JWT)
   - Token expiration and refresh mechanism
2. **Role-Based Access Control (RBAC)**
   - Admin role: Full system access, user management, configuration
   - Editor role: Document upload, search, download
   - Viewer role: Search and view only, no download
   - Custom roles per department or project
   - Permission inheritance from SharePoint groups
3. **Document-Level Permissions**
   - Inherit permissions from source system (SharePoint)
   - Filter search results based on user permissions
   - Audit trail for document access
   - Restricted document marking and handling

Data Security

Encryption:

- **In Transit**:
  - TLS 1.3 for all HTTP communications
  - Certificate pinning for API clients
  - HSTS (HTTP Strict Transport Security) headers
- **At Rest**:
  - AES-256 encryption for document storage
  - Encrypted Elasticsearch indices
  - Encrypted database backups
  - Key management using cloud provider KMS or on-premises HSM

Network Security:

- Virtual Private Cloud (VPC) with private subnets
- Security groups restricting access by IP and port
- Web Application Firewall (WAF) for DDoS protection
- Intrusion Detection System (IDS) monitoring

- Network segmentation between services

Compliance and Audit

Audit Logging:

- User authentication events (login, logout, failed attempts)
- Search queries with timestamp and user ID
- Document access (view, download) with IP address
- Administrative actions (user management, configuration changes)
- System events (service restarts, errors)
- Log retention: 1 year minimum, 7 years for regulated industries

Compliance Standards:

- SOC 2 Type II for service organizations
- ISO 27001 for information security management
- GDPR compliance for European data protection
- HIPAA compliance if handling health information
- Industry-specific standards (NERC CIP for energy, etc.)

Data Privacy:

- Personal information minimization in logs
- Right to erasure (delete user data on request)
- Data portability (export user data on request)
- Privacy policy and terms of service
- Consent management for analytics tracking

Known Limitations and Future Improvements

Current POC Limitations

1. **Manual Document Ingestion**
   - Limitation: Documents must be manually copied to folder and ingestion endpoint called
   - Impact: Not suitable for frequent document updates
   - Workaround: Schedule ingestion jobs using cron or Task Scheduler
   - Future: Automatic SharePoint synchronization with change detection
2. **No User Authentication**
   - Limitation: Anyone with network access can use system
   - Impact: Unsuitable for production deployment
   - Workaround: Deploy on internal network with VPN requirement
   - Future: SSO integration in Phase 1
3. **Single User Performance**
   - Limitation: Not tested with concurrent users
   - Impact: Performance under load unknown
   - Workaround: Deploy multiple backend instances with load balancer
   - Future: Load testing and horizontal scaling in Phase 1
4. **Limited Metadata Extraction**
   - Limitation: Only basic fields extracted (title, headings, content)
   - Impact: Cannot filter by date, author, project code

- Workaround: Encode metadata in filenames
- Future: Advanced metadata extraction in Phase 1

5. **Keyword Search Only**
    - Limitation: No semantic understanding of queries
    - Impact: May miss relevant documents with different terminology
    - Workaround: Use multiple query variations
    - Future: Semantic search in Phase 2
6. **No Document Upload UI**
    - Limitation: Documents added via file system only
    - Impact: Not user-friendly for non-technical users
    - Workaround: IT staff manages document additions
    - Future: Drag-and-drop upload interface in Phase 1

Planned Improvements

Short-term (Phase 1):

- SharePoint synchronization with scheduled jobs
- User authentication and authorization
- Document upload web interface
- Advanced filtering (date, type, project)
- Export search results to Excel
- Mobile-responsive design improvements

Medium-term (Phase 2):

- Semantic search with neural embeddings
- Automatic document categorization
- Document summarization
- Question answering interface
- Related document recommendations
- Multi-language support

Long-term (Phase 3):

- Analytics dashboard with insights
- Machine learning for result ranking
- Voice search capability
- Collaborative features (comments, annotations)
- Version control and change tracking
- Integration with CAD systems for engineering drawings

References

[1] Manning, C. D., Raghavan, P., & Schutze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

[2] Robertson, S., & Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval*, 3(4), 333-389.

[3] Elasticsearch B.V. (2023). *Elasticsearch Reference Guide*. https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html

[4] FastAPI Documentation. (2024). *FastAPI Framework*. https://fastapi.tiangolo.com/

[5] Next.js Documentation. (2024). *Next.js by Vercel*. https://nextjs.org/docs

[6] Docker Inc. (2024). *Docker Documentation*. https://docs.docker.com/

[7] Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of NAACL-HLT 2019*.

[8] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *Proceedings of EMNLP 2019*.

Appendix A: Glossary of Terms

API (Application Programming Interface): Set of protocols and tools for building software applications. Defines how components interact.

Async/Await: Programming pattern for handling asynchronous operations without blocking execution thread.

BM25: Best Matching 25 algorithm for ranking documents by relevance. Industry standard for full-text search.

CDN (Content Delivery Network): Distributed network of servers delivering web content to users based on geographic location.

CI/CD (Continuous Integration/Continuous Deployment): Automated process for integrating code changes and deploying to production.

Containerization: Packaging application with dependencies in isolated runtime environment for consistent deployment.

CORS (Cross-Origin Resource Sharing): Security mechanism allowing web applications to make requests to different domains.

Docker Compose: Tool for defining and running multi-container Docker applications using YAML configuration.

Elasticsearch: Distributed search and analytics engine built on Apache Lucene. Used for full-text search.

FastAPI: Modern Python web framework for building APIs with automatic documentation and type validation.

Fuzzy Matching: Search technique finding approximate matches allowing for spelling variations and typos.

Inverted Index: Data structure mapping terms to documents. Enables fast full-text search without scanning all documents.

JWT (JSON Web Token): Compact token format for securely transmitting information between parties as JSON object.

Next.js: React framework with server-side rendering, static generation, and built-in routing.

OAuth 2.0: Industry-standard protocol for authorization allowing third-party applications limited access.

pdfplumber: Python library for extracting text, tables, and metadata from PDF files.

REST (Representational State Transfer): Architectural style for distributed systems using HTTP methods and stateless communication.

SAML (Security Assertion Markup Language): XML-based standard for exchanging authentication and authorization data.

SSO (Single Sign-On): Authentication scheme allowing user to log in once and access multiple systems.

TLS (Transport Layer Security): Cryptographic protocol providing secure communication over computer network.

Uvicorn: Lightning-fast ASGI server for Python web applications with async support.

Vector Embedding: Dense numerical representation of text enabling semantic similarity computations.

VPC (Virtual Private Cloud): Isolated network environment within cloud infrastructure.

YAML (YAML Ain't Markup Language): Human-readable data serialization format used for configuration files.

Appendix B: Troubleshooting Guide

Issue: Elasticsearch fails to start

Symptoms: Service exits immediately after starting, logs show memory errors

Diagnosis Steps:

1. Check Docker Desktop memory allocation: Settings - Resources - Memory
2. View Elasticsearch logs: `docker logs woods-document-search-engine-elasticsearch-1`
3. Check for port conflicts: `netstat -an | findstr 9200`

Solutions:

- Increase Docker memory to minimum 4 GB
- Stop other services using port 9200
- Reduce Elasticsearch heap size in docker-compose.yml: ES_JAVA_OPTS=-Xms256m -Xmx256m

Issue: Search returns no results

Symptoms: All searches return empty results array, total count is 0

Diagnosis Steps:

1. Verify Elasticsearch is running: `curl http://localhost:9200`
2. Check index exists: `curl http://localhost:9200/wood_ai_documents`
3. Check document count: `curl http://localhost:9200/wood_ai_documents/_count`
4. View backend logs: `docker logs woods-document-search-engine-backend-1`

Solutions:

- Run ingestion endpoint: `curl -X POST http://localhost:8000/ingest`
- Verify documents folder contains PDF or DOCX files
- Check file permissions on documents folder
- Recreate index: Delete index, restart services, re-ingest

Issue: PDF preview shows error

Symptoms: View Details button opens modal but shows "Document not found" error

Diagnosis Steps:

1. Check document ID in browser developer tools Network tab
2. Verify file exists: `docker exec woods-document-search-engine-backend-1 ls /app/documents`
3. Check backend logs for file matching attempts

Solutions:

- Re-run ingestion to ensure consistent document IDs
- Verify filename matches document ID from search results
- Check for special characters in filenames (rename if necessary)

Issue: Frontend cannot connect to backend

Symptoms: Search button shows loading indefinitely, browser console shows connection errors

Diagnosis Steps:

1. Verify backend is running: `docker ps`
2. Check backend health: `curl http://localhost:8000/health`
3. View browser console for exact error message
4. Check CORS configuration in backend

Solutions:

- Ensure both frontend and backend containers are running
- Verify NEXT_PUBLIC_API_URL environment variable is set correctly
- Check browser is not blocking requests (disable ad blockers)
- Restart frontend container: `docker-compose restart frontend`

Document Revision History

Version	Date	Author	Changes
1.0	2025-11-30	Aaron Sequeira	Initial POC documentation

END OF DOCUMENT