

Vector Knowledge Base

A Semantic Search Engine for Personal Document Management

i3T4AN (Ethan Blair)

December 2025

Abstract

Personal knowledge management has become increasingly challenging as digital document collections grow in size and complexity. Traditional keyword-based search methods fail to capture the semantic relationships between concepts, often returning irrelevant results or missing semantically similar content that uses different terminology. This paper presents **Vector Knowledge Base**, a semantic search engine designed for personal document management that leverages modern natural language processing techniques to enable meaning-based retrieval.

The system employs the `all-mpnet-base-v2` transformer model from the SentenceTransformers library [1] to generate 768-dimensional dense vector embeddings from document content. These embeddings are stored in Qdrant, a high-performance vector database, enabling cosine similarity-based semantic search. The document processing pipeline supports 23 file formats through 9 specialized extractors, including PDF, Microsoft Office documents, images (via OCR), and source code files with AST-aware chunking.

Key features include: (1) HDBSCAN density-based clustering [2] with automatic semantic naming via TF-IDF, (2) interactive 3D visualization of the embedding space using PCA for dimensionality reduction, (3) a hierarchical folder organization system with batch upload capabilities, and (4) Model Context Protocol (MCP) integration for AI agent connectivity.

The system achieves sub-50ms search latency for collections under 10,000 vectors and scales efficiently to 100,000+ documents. GPU acceleration via CUDA (NVIDIA) and MPS (Apple Silicon) provides 5-10x speedup for embedding generation. The complete solution is containerized using Docker for reproducible deployment, with a FastAPI backend and vanilla JavaScript frontend.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	The Promise of Semantic Search	5
1.3	System Overview	5
1.4	Contributions	5
1.5	Paper Organization	6
2	Background and Related Work	6
2.1	Vector Databases	6
2.1.1	Vector Embeddings	6
2.1.2	Qdrant Vector Database	6
2.1.3	Comparison with Alternatives	7
2.2	Embedding Models	7
2.2.1	Transformer Architecture	7
2.2.2	SentenceTransformers	7
2.3	Clustering Algorithms	8
2.3.1	Why Not K-Means?	8
2.3.2	HDBSCAN	8
2.3.3	Semantic Cluster Naming with TF-IDF	9
2.4	Dimensionality Reduction	9
2.4.1	Principal Component Analysis (PCA)	9
2.4.2	PCA vs. Alternative Methods	9
3	System Architecture	10
3.1	High-Level Architecture	10
3.1.1	Architecture Overview	10
3.1.2	Service Configuration	10
3.2	Backend Components	11
3.2.1	Module Overview	11
3.2.2	Async Design Patterns	11
3.2.3	Module Dependencies	11
3.3	Frontend Components	12
3.3.1	HTML Pages	12
3.3.2	JavaScript Modules	12
3.3.3	Modular CSS Architecture	13
3.3.4	3D Visualization with Three.js	13
3.4	Database Layer	13
3.4.1	Qdrant Vector Database	13
3.4.2	SQLite Metadata Database	14
3.4.3	Document Registry	14
3.4.4	Database Schema Overview	14
4	Document Processing Pipeline	15
4.1	File Extraction	15
4.1.1	Pipeline Overview	15
4.1.2	Factory Pattern	15
4.1.3	Extractor Classes	15
4.2	Text Chunking	16
4.2.1	Chunking Parameters	16
4.2.2	Prose Chunking	16

4.2.3	AST-Aware Python Chunking	17
4.3	Embedding Generation	17
4.3.1	Singleton Pattern	17
4.3.2	GPU Acceleration	18
4.4	Vector Storage	18
4.4.1	Batch-Embed-Then-Upsert Architecture	18
4.4.2	Batch Upsert with Chunking	19
4.4.3	Payload Metadata	19
5	Search and Retrieval	20
5.1	Semantic Search	20
5.1.1	Search Architecture	20
5.1.2	Search Endpoint	20
5.2	Filtering Capabilities	21
5.3	Performance Optimization	21
6	Clustering and Organization	21
6.1	HDBSCAN Clustering	21
6.1.1	Algorithm Parameters	21
6.2	Semantic Cluster Naming	22
6.2.1	Naming Algorithm	22
6.2.2	Clustering Workflow	23
7	3D Visualization	23
7.1	Dimensionality Reduction	23
7.1.1	Cold-Start Robustness	23
7.2	Interactive Visualization	24
7.2.1	Three.js Scene Setup	24
7.2.2	Features	24
7.2.3	Visualization Screenshot	25
8	File Management System	26
8.1	Folder Organization	26
8.1.1	SQLite Hierarchy	26
8.1.2	Breadcrumb Navigation	26
8.1.3	Drag-and-Drop	26
8.2	Batch Upload	26
8.2.1	Folder Structure Preservation	26
8.2.2	Progress Tracking	26
8.2.3	File Organization Screenshot	27
8.3	Data Export	27
9	API Design	27
9.1	RESTful Endpoints	27
9.1.1	Core Endpoints	27
9.1.2	Pydantic Models	28
9.2	Model Context Protocol (MCP)	28
9.2.1	MCP Architecture	28
9.2.2	MCP Tools	29
9.2.3	Claude Desktop Integration	29
10	Deployment	30

10.1 Docker Deployment	30
10.1.1 Docker Compose Configuration	30
10.1.2 Startup	31
10.2 Performance Mode (GPU)	31
10.2.1 Native Setup	31
10.2.2 Deployment Comparison	31
11 Security	31
11.1 Rate Limiting	32
11.1.1 SlowAPI Integration	32
11.1.2 Configurable Limits	32
11.2 Admin Protection	32
11.2.1 ADMIN_KEY Authentication	32
11.2.2 CORS Configuration	32
11.3 Input Sanitization	32
12 Performance Analysis	33
12.1 Benchmarks	33
12.1.1 Performance Metrics	33
12.1.2 GPU Acceleration Impact	33
12.2 Scalability	33
12.2.1 Vector Storage Capacity	33
12.2.2 O(1) Document Listing	34
12.2.3 3D Cache Strategy	34
13 Conclusion	34
13.1 Summary of Contributions	34
13.2 Future Work	34
13.3 Final Remarks	35
A Configuration Reference	37
B API Endpoint Quick Reference	37
C Supported File Types	38
D User Interface Screenshots	39

1 Introduction

1.1 Motivation

In the modern digital era, individuals accumulate vast collections of documents spanning research papers, lecture notes, code repositories, project documentation, and personal writings. The challenge of efficiently retrieving specific information from these heterogeneous collections has become a significant productivity bottleneck. Traditional file system organization and keyword-based search methods, while familiar, suffer from fundamental limitations that reduce their effectiveness for knowledge work.

Keyword search operates on exact lexical matching—a query for “machine learning” will not find documents discussing “neural networks” or “deep learning” unless those exact terms appear in the text. This limitation, known as the *vocabulary mismatch problem*, means that semantically relevant documents are frequently missed. Furthermore, users must recall the specific terminology used in the original document, which is cognitively demanding and error-prone.

1.2 The Promise of Semantic Search

Semantic search addresses these limitations by operating on the *meaning* of text rather than its surface form. By representing documents and queries as dense vector embeddings in a high-dimensional semantic space, we can measure similarity based on conceptual proximity rather than lexical overlap. Documents discussing related concepts cluster together in this space, enabling retrieval based on semantic relevance.

The emergence of transformer-based language models [3,4] has dramatically improved the quality of text embeddings. Models like Sentence-BERT [1] are specifically trained to produce embeddings where semantically similar sentences have high cosine similarity, making them ideal for information retrieval applications.

1.3 System Overview

Vector Knowledge Base is a complete semantic search solution designed for personal document management. The system comprises three main components:

1. **Document Processing Pipeline:** Extracts text from 23 file formats using 9 specialized extractors, intelligently chunks content, and generates vector embeddings using the `all-mpnet-base-v2` model.
2. **Vector Search Engine:** Stores embeddings in Qdrant vector database and performs cosine similarity-based retrieval with support for metadata filtering.
3. **Knowledge Organization:** Applies HDBSCAN clustering to automatically group related documents, generates semantic cluster names using TF-IDF, and provides 3D visualization of the embedding space.

The system is built with a FastAPI backend providing RESTful endpoints and a vanilla JavaScript frontend with no framework dependencies. Docker containerization ensures reproducible deployment across environments.

1.4 Contributions

This paper makes the following contributions:

- A complete, open-source semantic search system optimized for personal document collections

- A modular document extraction architecture using the Factory design pattern with support for 23 file formats
- Integration of HDBSCAN clustering with TF-IDF-based semantic naming for automatic knowledge organization
- Interactive 3D visualization of high-dimensional embedding spaces using PCA and Three.js
- Model Context Protocol (MCP) integration enabling AI agent access to the knowledge base
- Comprehensive deployment options including Docker and native GPU-accelerated modes

1.5 Paper Organization

The remainder of this paper is organized as follows. Section 2 provides background on vector databases, embedding models, and clustering algorithms. Section 3 describes the system architecture. Section 4 details the document processing pipeline. Section 5 covers search and retrieval. Section 6 explains the clustering and organization features. Section 7 presents the 3D visualization system. Section 8 describes file management capabilities. Section 9 documents the API design. Sections 10 and 11 cover deployment and security considerations. Section 12 presents performance analysis, and Section 13 concludes.

2 Background and Related Work

This section provides essential background on the core technologies underlying the Vector Knowledge Base system.

2.1 Vector Databases

Vector databases are specialized storage systems designed for efficient similarity search over high-dimensional vector data. Unlike traditional relational databases that excel at exact matching and structured queries, vector databases optimize for approximate nearest neighbor (ANN) search, enabling retrieval of the most similar vectors to a given query vector.

2.1.1 Vector Embeddings

A vector embedding is a numerical representation of data—typically text—as a point in a high-dimensional space. The key property of useful embeddings is that semantically similar items are mapped to nearby points in the vector space. For text, this means sentences with similar meanings have vectors with high cosine similarity:

$$\text{cosine_similarity}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| \cdot |\mathbf{b}|} \quad (1)$$

Modern embedding models produce dense vectors with hundreds of dimensions. The `all-mpnet-base-v2` model used in this system produces 768-dimensional vectors.

2.1.2 Qdrant Vector Database

We selected Qdrant [5] as our vector database for several reasons:

- **High Performance:** Written in Rust for maximum efficiency, Qdrant delivers sub-millisecond search latency for collections of reasonable size.

- **Rich Filtering:** Supports filtering search results by payload metadata, enabling queries like “find similar documents from this date range” or “within this cluster.”
- **Async API:** Provides a native async Python client that integrates seamlessly with FastAPI’s async event loop, preventing blocking operations.
- **Scalability:** Supports horizontal scaling through sharding and replication for production workloads.
- **Ease of Deployment:** Available as a Docker image, making it trivial to deploy alongside the application.

Qdrant uses the HNSW (Hierarchical Navigable Small World) algorithm for approximate nearest neighbor search, providing a favorable trade-off between search accuracy and speed.

2.1.3 Comparison with Alternatives

Several other vector databases exist in the market [6]:

- **Weaviate:** Offers built-in vectorization but adds complexity for our use case where we control the embedding model.
- **Milvus:** Enterprise-focused with more operational overhead than needed for personal use.
- **Pinecone:** Cloud-hosted SaaS that requires internet connectivity and subscription costs.
- **Elasticsearch** [7]: While it now supports vector search, it remains primarily optimized for traditional keyword search (BM25) rather than vector operations.

Qdrant’s combination of performance, simplicity, and open-source availability made it the optimal choice for a self-hosted personal knowledge base.

2.2 Embedding Models

2.2.1 Transformer Architecture

The transformer architecture [3] revolutionized natural language processing by replacing recurrent structures with self-attention mechanisms. The key innovation is the attention mechanism, which computes weighted combinations of all input positions:

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (2)$$

where Q , K , and V are query, key, and value matrices derived from the input, and d_k is the dimension of the keys. This allows the model to capture long-range dependencies without the sequential processing bottleneck of RNNs.

BERT (Bidirectional Encoder Representations from Transformers) [4] applied this architecture to produce contextualized word embeddings by training on masked language modeling and next sentence prediction tasks.

2.2.2 SentenceTransformers

While BERT produces excellent token-level embeddings, directly averaging these for sentence-level representations yields suboptimal results for semantic similarity tasks. Sentence-BERT [1] addresses this by fine-tuning BERT using siamese and triplet network structures on sentence pairs with known similarity relationships.

The SentenceTransformers library [8] provides pre-trained models optimized for various tasks. We use `all-mpnet-base-v2`, which:

- Is trained on over 1 billion sentence pairs
- Produces 768-dimensional embeddings
- Achieves state-of-the-art performance on semantic similarity benchmarks
- Supports sequences up to 384 tokens

The embedding generation process in our system:

1. Load the model once at startup (singleton pattern)
2. Automatically detect GPU availability (CUDA, MPS, or CPU fallback)
3. Process text through the model's `encode()` method
4. Return embeddings as Python lists for storage

2.3 Clustering Algorithms

Automatic organization of documents requires clustering algorithms that can identify natural groupings within the embedding space.

2.3.1 Why Not K-Means?

K-means clustering, while widely used, has significant limitations for document clustering:

- Requires specifying the number of clusters k in advance
- Assumes spherical clusters of similar size
- Cannot identify noise points or outliers
- Sensitive to initialization

In a personal knowledge base, the number and shape of topic clusters is unknown and may vary dramatically as documents are added. We need an algorithm that can automatically determine the cluster structure.

2.3.2 HDBSCAN

HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) [2] addresses these limitations:

- **Automatic Cluster Detection:** Determines the number of clusters from the data density structure, with no need to specify k .
- **Arbitrary Cluster Shapes:** Works with non-spherical clusters of varying densities.
- **Noise Handling:** Explicitly identifies outlier points that don't belong to any cluster (labeled as -1).
- **Hierarchical:** Produces a hierarchy of clusters, with the "Excess of Mass" (EOM) method selecting the most stable clusters.

Our implementation uses the following parameters:

- `min_cluster_size=5`: Minimum points to form a cluster
- `min_samples=3`: Controls how conservative clustering is

- `metric='euclidean'`: Distance metric (cosine is also supported)
- `cluster_selection_method='eom'`: Selects stable clusters

2.3.3 Semantic Cluster Naming with TF-IDF

After clustering, we generate human-readable names for each cluster using TF-IDF (Term Frequency-Inverse Document Frequency):

$$\text{TF-IDF}(t, d, D) = \text{tf}(t, d) \times \log \left(\frac{|D|}{|\{d \in D : t \in d\}|} \right) \quad (3)$$

For each cluster, we:

1. Collect all text chunks belonging to that cluster
2. Compute TF-IDF scores across the cluster corpus
3. Extract the top 3 most distinctive terms or phrases (1-2 word n-grams)
4. Combine them into a descriptive cluster name

This produces names like “Machine Learning & Neural Networks” or “Python & Data Processing” that meaningfully describe cluster contents.

2.4 Dimensionality Reduction

Visualizing 768-dimensional embedding vectors requires projecting them to a viewable 2D or 3D space.

2.4.1 Principal Component Analysis (PCA)

PCA is a linear dimensionality reduction technique that finds the directions of maximum variance in the data. Given a data matrix X , PCA computes the eigenvectors of the covariance matrix $X^T X$ and projects data onto the top k eigenvectors (principal components).

For our 3D visualization, we reduce from 768 to 3 dimensions:

$$\mathbf{z} = W^T \mathbf{x} \quad (4)$$

where $W \in \mathbb{R}^{768 \times 3}$ contains the top 3 principal components and \mathbf{x} is the original embedding.

2.4.2 PCA vs. Alternative Methods

We chose PCA over alternatives like t-SNE or UMAP for several reasons:

- **Speed:** PCA is computationally efficient, important for real-time visualization updates.
- **Deterministic:** Produces consistent results across runs (unlike stochastic methods).
- **New Point Projection:** Can efficiently project new query embeddings without refitting, using the stored transformation matrix.
- **Global Structure:** Preserves global relationships, which matters for understanding overall knowledge organization.

While t-SNE and UMAP may produce visually appealing local clusters, their stochastic nature and inability to project new points without refitting makes them less suitable for our interactive use case. However, the architecture is extensible: `dimensionality_reduction.py` includes hooks to support UMAP if the `umap-learn` library is installed, falling back to PCA otherwise. This allows users requiring non-linear manifold visualizations to enable UMAP without code changes.

3 System Architecture

This section describes the overall architecture of the Vector Knowledge Base system, including its three-tier design, component breakdown, and data storage strategy.

3.1 High-Level Architecture

The system follows a classic three-tier architecture: presentation (frontend), business logic (backend), and data storage (databases). All components are containerized using Docker for reproducible deployment.

3.1.1 Architecture Overview

Figure 1 illustrates the high-level system architecture.

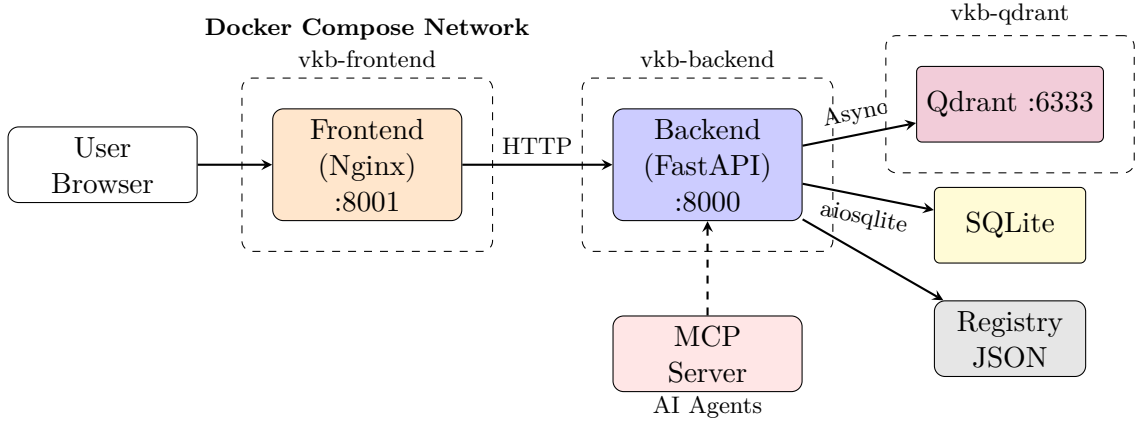


Figure 1: High-level system architecture showing the three-tier design with Docker containerization.

3.1.2 Service Configuration

The system deploys three Docker containers orchestrated by `docker-compose.yml`:

1. **vkb-qdrant**: The Qdrant vector database (official image), exposed on port 6333. Stores vector embeddings with cosine similarity search.
2. **vkb-backend**: FastAPI application built from custom Dockerfile, exposed on port 8000. Handles all business logic, document processing, and API endpoints.
3. **vkb-frontend**: Nginx Alpine serving static files, exposed on port 8001. Provides the user interface with no server-side rendering.

All containers communicate over a shared Docker bridge network (`vkb-network`), with the backend connecting to Qdrant via the container name (`qdrant:6333`).

3.2 Backend Components

The backend is implemented in Python using FastAPI [9], an async-first web framework. The codebase consists of 14 Python modules organized by responsibility.

3.2.1 Module Overview

Table 1 describes each backend module:

Table 1: Backend Python modules and their responsibilities.

Module	Responsibility	LOC
<code>main.py</code>	FastAPI application, all endpoints	1,200+
<code>vector_db.py</code>	Qdrant async client wrapper	300
<code>embedding_service.py</code>	SentenceTransformer embedding	90
<code>clustering.py</code>	HDBSCAN clustering + TF-IDF naming	150
<code>dimensionality_reduction.py</code>	PCA for 3D projection	190
<code>chunker.py</code>	Text chunking (prose + code-aware)	230
<code>ingestion.py</code>	Document processing orchestration	200
<code>document_registry.py</code>	O(1) document listing (JSON)	220
<code>filesystem_db.py</code>	Folder hierarchy (SQLite)	240
<code>config.py</code>	Pydantic settings, device detection	130
<code>constants.py</code>	Shared constants	15
<code>exceptions.py</code>	Custom exception classes	35
<code>jobs.py</code>	Background job tracking	120
<code>mcp_server.py</code>	Model Context Protocol server	100

Additionally, the `extractors/` package contains 9 file-type-specific extractors plus the factory pattern implementation (12 files total).

3.2.2 Async Design Patterns

FastAPI’s async support is leveraged throughout the codebase to prevent blocking the event loop:

- **Async Qdrant Client:** All vector database operations use `AsyncQdrantClient`, enabling non-blocking search and upsert operations.
- **aiosqlite:** The SQLite database for folder management uses `aiosqlite` for async I/O.
- **ThreadPoolExecutor:** CPU-intensive operations like embedding generation run in a thread pool via `loop.run_in_executor()`, keeping the main event loop responsive.

3.2.3 Module Dependencies

Figure 2 shows the dependency relationships between major backend modules.

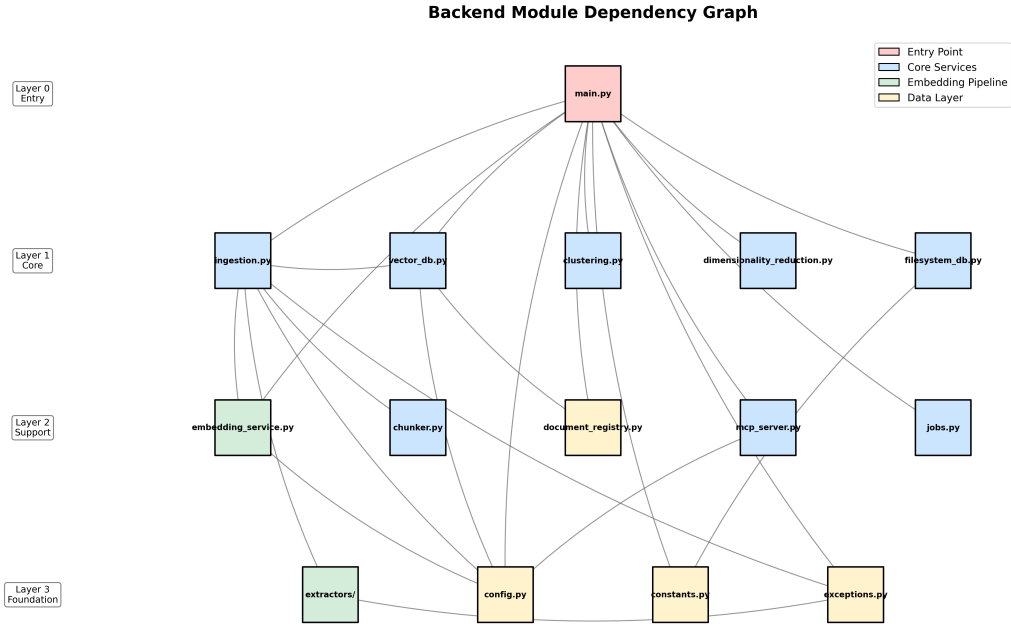


Figure 2: Backend module dependency graph. Colors indicate: entry point (red), core services (blue), embedding pipeline (green), and data layer (yellow). Arrows indicate import relationships.

3.3 Frontend Components

The frontend is implemented as a multi-page application (MPA) using component-based vanilla ES6 JavaScript with no framework dependencies. While no framework is used, the code is structured into reusable ES6 classes (e.g., `FileSystem`, `NotificationSystem`, `EmbeddingVisualizer`) rather than loose scripts, ensuring maintainability. This design choice prioritizes simplicity and reduces bundle size.

3.3.1 HTML Pages

The application consists of three main pages:

- **index.html:** The main search interface. Users enter semantic queries, view results with similarity scores, and optionally filter by cluster.
- **documents.html:** Document management page. Users can upload files (single or batch), view all uploaded documents, and delete documents.
- **files.html:** Folder organization page. Users can create hierarchical folders, drag-and-drop files between folders, and manage document organization.

3.3.2 JavaScript Modules

Eight JavaScript files handle frontend logic:

Table 2: Frontend JavaScript modules.

File	Functionality
<code>search.js</code>	Semantic search, result rendering, cluster filtering
<code>upload.js</code>	File uploads, batch processing, progress tracking
<code>documents.js</code>	Document listing, deletion
<code>filesystem.js</code>	Folder creation, drag-and-drop, tree navigation
<code>notifications.js</code>	Toast notifications system
<code>embedding-visualizer.js</code>	Three.js 3D visualization (21KB)
<code>config.js</code>	API base URL configuration
<code>constants.js</code>	Shared constants (folder IDs, limits)

3.3.3 Modular CSS Architecture

Styles are organized into 7 focused CSS files for maintainability:

- `base.css`: CSS variables, typography, resets
- `layout.css`: Page structure, grid, flexbox utilities
- `components.css`: Buttons, cards, inputs, badges
- `modals.css`: Modal dialogs and overlays
- `filesystem.css`: Folder tree, file cards, breadcrumbs
- `batch-upload.css`: Upload progress, folder preview
- `animations.css`: Transitions, skeleton loaders

3.3.4 3D Visualization with Three.js

The `embedding-visualizer.js` module (21KB) provides interactive 3D visualization of the embedding space:

- **Rendering**: WebGL-based rendering via Three.js with orbit controls for rotation, zoom, and pan.
- **Point Cloud**: Each document chunk is rendered as a colored sphere, with colors assigned by cluster.
- **Query Highlighting**: When a search is performed, the query embedding is projected and displayed as a distinct marker, with lines connecting to the most similar results.
- **Interactivity**: Hovering over points shows document metadata; clicking navigates to the document.

3.4 Database Layer

The system uses a dual-database architecture: Qdrant for vector storage and SQLite for relational metadata. A JSON-based document registry provides $O(1)$ document listing.

3.4.1 Qdrant Vector Database

Qdrant stores the core vector embeddings with the following configuration:

- **Collection**: `vector_db` (configurable via `QDRANT_COLLECTION`)

- **Vector Size:** 768 dimensions (matching `all-mpnet-base-v2`)
- **Distance Metric:** Cosine similarity
- **Payload:** Each vector includes metadata (filename, text, chunk index, upload date, cluster ID)

3.4.2 SQLite Metadata Database

SQLite (`metadata.db`) stores the folder hierarchy using two tables:

```
1 CREATE TABLE folders (
2     id TEXT PRIMARY KEY,
3     name TEXT NOT NULL,
4     parent_id TEXT REFERENCES folders(id) ON DELETE CASCADE
5 );
6
7 CREATE TABLE file_folders (
8     document_id TEXT PRIMARY KEY,
9     filename TEXT NOT NULL,
10    folder_id TEXT REFERENCES folders(id) ON DELETE SET NULL
11 );
```

Listing 1: SQLite schema for folder management.

The **folders** table supports arbitrary nesting depth. When a folder is deleted, its files become “unsorted” (via `ON DELETE SET NULL`).

3.4.3 Document Registry

The `DocumentRegistry` class maintains a JSON file (`data/documents.json`) for $O(1)$ document listing:

- **Problem:** Listing all documents from Qdrant requires scrolling through all vectors and deduplicating by `document_id`—an $O(n)$ operation.
- **Solution:** Maintain a separate registry mapping `document_id` to metadata (filename, upload date, chunk count).
- **Self-Healing Sync:** If the JSON registry is corrupted or deleted, the system automatically detects the mismatch and rebuilds the registry by crawling the Vector DB via `sync_from_qdrant()`. This provides fault tolerance and ensures data consistency.

3.4.4 Database Schema Overview

Figure 3 shows the overall data storage architecture.

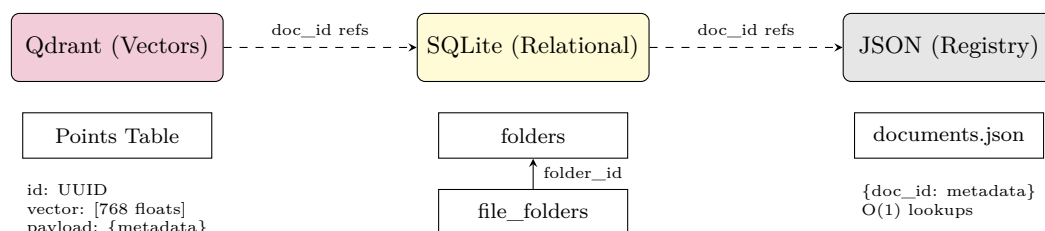


Figure 3: Database schema showing Qdrant vectors, SQLite folder hierarchy, and JSON document registry.

4 Document Processing Pipeline

This section describes the complete pipeline for processing documents from upload through to vector storage. The pipeline consists of four stages: extraction, chunking, embedding, and storage.

4.1 File Extraction

The system supports 23 file formats through 9 specialized extractor classes. A factory pattern enables modular, extensible extraction.

4.1.1 Pipeline Overview

Figure 4 illustrates the document processing pipeline.

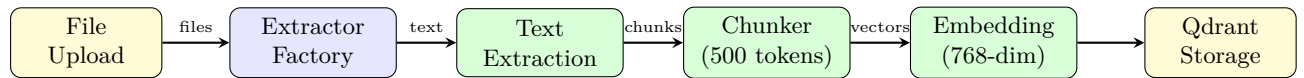


Figure 4: Document processing pipeline from upload to vector storage.

4.1.2 Factory Pattern

The `ExtractorFactory` class maps file extensions to extractor implementations, enabling easy addition of new formats without modifying client code.

```
1 class ExtractorFactory:
2     _extractors: Dict[str, Type[BaseExtractor]] = {
3         '.pdf': PDFExtractor,
4         '.docx': DocxExtractor,
5         '.pptx': PptxExtractor,
6         '.jpg': ImageExtractor, # Also: .jpeg, .png, .webp
7         '.txt': TextExtractor, # Also: .md
8         '.py': CodeExtractor, # Also: .js, .java, .cpp, .html, .css, .json
9         '.cs': CsExtractor,
10        '.xlsx': XlsxExtractor,
11        '.csv': CsvExtractor
12    }
13
14    @classmethod
15    def get_extractor(cls, file_path: str) -> BaseExtractor:
16        _, ext = os.path.splitext(file_path)
17        ext = ext.lower()
18        extractor_class = cls._extractors.get(ext)
19        if not extractor_class:
20            raise InvalidFileFormatError(f"Unsupported: {ext}")
21        return extractor_class()
```

Listing 2: `ExtractorFactory.get_extractor()` from `factory.py`.

4.1.3 Extractor Classes

Table 3 lists all 9 extractor classes with their supported extensions and underlying libraries.

Table 3: File extractors with supported extensions and libraries.

Extractor	Extensions	Library	Notes
PDFExtractor	.pdf	pypdf	Text extraction
DocxExtractor	.docx	docx2txt	Word documents
PptxExtractor	.pptx, .ppt	python-pptx	Slide extraction
ImageExtractor	.jpg, .jpeg, .png, .webp	pytesseract	OCR (grayscale preprocessing)
TextExtractor	.txt, .md	built-in	Plain text
CodeExtractor	.py, .js, .java, .cpp, .html, .css, .json, .xml, .yaml, .yml	built-in	Source code
CsExtractor	.cs	custom regex	C# parsing
XlsxExtractor	.xlsx	openpyxl	Pipe-linearized rows
CsvExtractor	.csv	built-in csv	Pipe-linearized rows

4.2 Text Chunking

After extraction, text is split into chunks suitable for embedding. The **Chunker** class implements intelligent chunking with two strategies: prose-aware for documents and AST-aware for code.

4.2.1 Chunking Parameters

- **Chunk Size:** 500 tokens (configurable via `CHUNK_SIZE`)
- **Overlap:** 50 tokens (configurable via `CHUNK_OVERLAP`)
- **Max Model Tokens:** 500 (buffer below 512 model limit)

The tokenizer from `sentence-transformers/all-mpnet-base-v2` is used for accurate token counting.

4.2.2 Prose Chunking

For prose text, the chunker respects sentence boundaries to avoid splitting mid-sentence:

```

1 def chunk_text(self, text: str, metadata: Optional[Dict] = None) -> List[Dict]:
2     """Split text into chunks. Detects code vs prose."""
3     if not text:
4         return []
5     metadata = metadata or {}
6     is_code = metadata.get("language") in ["py", "python", "js", "javascript",
7     "java", "cpp"]
8
9     if is_code and metadata.get("language") in ["py", "python"]:
10         return self._chunk_python_code(text, metadata)
11     else:
12         return self._chunk_prose(text, metadata)

```

Listing 3: Chunker.chunk_text() dispatch logic from chunker.py.

The prose chunking algorithm:

1. Split text into sentences using regex: `(?<=[.!?])\s+(?=[A-Z])`
2. Pre-compute token counts for all sentences (cached)
3. Build chunks by accumulating sentences until `chunk_size` exceeded
4. Apply overlap by rewinding to include trailing sentences in next chunk

4.2.3 AST-Aware Python Chunking

For Python files, the chunker uses the `ast` module to respect function and class boundaries:

```
1 def _chunk_python_code(self, text: str, metadata: Dict) -> List[Dict]:
2     """Chunk Python code respecting class/function boundaries using AST."""
3     chunks = []
4     try:
5         tree = ast.parse(text)
6         lines = text.splitlines()
7         current_chunk_lines = []
8         current_chunk_token_count = 0
9
10        for node in tree.body:
11            if hasattr(node, 'lineno') and hasattr(node, 'end_lineno'):
12                start = node.lineno - 1
13                end = node.end_lineno
14                node_lines = lines[start:end]
15                node_text = "\n".join(node_lines)
16                node_tokens = self._count_tokens(node_text)
17
18                if current_chunk_token_count + node_tokens > self.chunk_size:
19                    if current_chunk_lines:
20                        chunk_text = "\n".join(current_chunk_lines)
21                        chunks.append({
22                            "text": chunk_text,
23                            "chunk_index": len(chunks),
24                            "token_count": current_chunk_token_count,
25                            "metadata": metadata.copy()
26                        })
27                        current_chunk_lines = []
28                        current_chunk_token_count = 0
29
30                    current_chunk_lines.extend(node_lines)
31                    current_chunk_token_count += node_tokens
32                # ... add remaining chunk
33    except SyntaxError:
34        return self._chunk_prose(text, metadata) # Fallback
35    return chunks
```

Listing 4: `Chunker._chunk_python_code()` from `chunker.py`.

This ensures that function definitions and class bodies are kept together when possible, improving embedding quality for code search.

4.3 Embedding Generation

The `EmbeddingService` class generates 768-dimensional vector embeddings using Sentence-Transformers.

4.3.1 Singleton Pattern

The model is loaded once and reused across requests to avoid repeated loading overhead:

```
1 class EmbeddingService:
2     _instance = None
3     _model = None
4     _executor = ThreadPoolExecutor(max_workers=2) # For async
5
6     def __new__(cls):
7         if cls._instance is None:
8             cls._instance = super(EmbeddingService, cls).__new__(cls)
9         return cls._instance
```

```

10
11 def _load_model(self):
12     """Load model with GPU support if available"""
13     logger.info(f"Loading model: {settings.EMBEDDING_MODEL}")
14     logger.info(f"Using device: {settings.DEVICE}")
15     self._model = SentenceTransformer(
16         settings.EMBEDDING_MODEL,
17         device=settings.DEVICE
18     )
19     # Log GPU memory if CUDA
20     if settings.DEVICE == "cuda":
21         import torch
22         mem = torch.cuda.get_device_properties(0).total_memory / 1024**3
23         logger.info(f"CUDA GPU Memory: {mem:.1f} GB")
24     elif settings.DEVICE == "mps":
25         logger.info("MPS (Metal) acceleration enabled")
26
27     async def embed_batch_async(self, texts: List[str]) -> List[List[float]]:
28         """Async batch embedding via ThreadPoolExecutor"""
29         loop = asyncio.get_event_loop()
30         return await loop.run_in_executor(self._executor, self.embed_batch,
        texts)

```

Listing 5: EmbeddingService with GPU support from embedding_service.py.

4.3.2 GPU Acceleration

Device detection is automatic via `config.py`:

- **MPS (Apple Silicon):** Detected via `torch.backends.mps.is_available()`
- **CUDA (NVIDIA):** Detected via `torch.cuda.is_available()`
- **CPU Fallback:** Used when no GPU is available

GPU acceleration provides 5-10x speedup for embedding generation, particularly beneficial during batch uploads.

4.4 Vector Storage

The `VectorDBClient` class wraps the Qdrant async client, providing high-level operations for vector management.

4.4.1 Batch-Embed-Then-Upsert Architecture

The batch upload endpoint (`/upload-batch`) implements a highly optimized multi-stage pipeline for processing multiple files from the same folder:

1. **Batch Extraction/Chunking:** All files are processed sequentially through extraction and chunking, collecting text chunks without embedding.
2. **Batch Embedding:** All collected text chunks are sent to the GPU/CPU in a single batch operation via `embed_batch_async()`, maximizing hardware utilization.
3. **Batch Upsert:** Vectors are pushed to Qdrant in chunks of 500 to prevent timeouts while maintaining throughput.
4. **Single Folder Resolution:** The folder hierarchy is resolved once per batch rather than per-file, reducing SQLite queries.

This architecture significantly improves upload speed compared to processing files individually, as embedding and database operations benefit from batching.

4.4.2 Batch Upsert with Chunking

Large batch uploads can timeout if sent in a single request. The `upsert_batch()` method splits points into chunks of 500:

```
1 async def upsert_batch(self, points: List[Dict]) -> None:
2     """Upsert points in chunks to avoid timeouts."""
3     if not points:
4         return
5
6     logger.info(f"Batch upserting {len(points)} vectors")
7     from qdrant_client.models import PointStruct
8
9     CHUNK_SIZE = 500 # Avoid timeout on large batches
10    total_points = len(points)
11
12    for i in range(0, total_points, CHUNK_SIZE):
13        chunk = points[i:i + CHUNK_SIZE]
14        chunk_num = (i // CHUNK_SIZE) + 1
15        total_chunks = (total_points + CHUNK_SIZE - 1) // CHUNK_SIZE
16        logger.info(f"Upserting chunk {chunk_num}/{total_chunks}")
17
18        qdrant_points = [
19            PointStruct(
20                id=point['id'],
21                vector=point['vector'],
22                payload=point['payload']
23            )
24            for point in chunk
25        ]
26
27        await self.client.upsert(
28            collection_name=self.collection_name,
29            points=qdrant_points,
30            wait=True # Ensure completion before proceeding
31        )
32
33    logger.info(f"Successfully upserted {total_points} vectors")
```

Listing 6: `VectorDBClient.upsert_batch()` from `vector_db.py`.

4.4.3 Payload Metadata

Each vector is stored with a payload containing:

- `document_id`: Unique identifier for the source document
- `filename`: Original filename
- `text`: The chunk text (for display in search results)
- `chunk_index`: Position within the document
- `total_chunks`: Total chunks in the document
- `upload_date`: Unix timestamp
- `cluster_id`: Assigned after clustering (-1 for unclustered)

5 Search and Retrieval

The search system transforms natural language queries into vector embeddings and finds the most semantically similar document chunks using cosine similarity.

5.1 Semantic Search

5.1.1 Search Architecture

Figure 5 illustrates the semantic search pipeline.

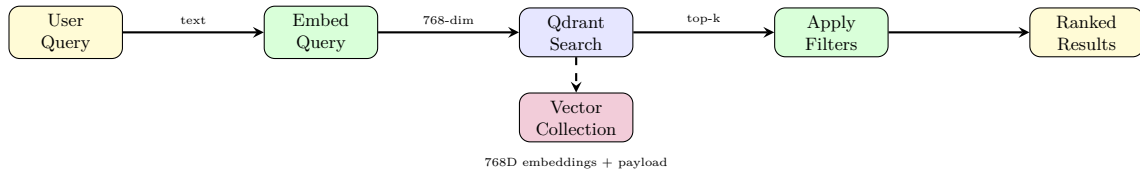


Figure 5: Semantic search pipeline from user query to ranked results.

5.1.2 Search Endpoint

The `/search` endpoint implements a three-step process:

```
1 @app.post("/search", response_model=SearchResponse)
2 @limiter.limit(settings.RATE_LIMIT_SEARCH)
3 async def search_documents(request: Request, search_req: SearchRequest):
4     """Search for documents using vector similarity."""
5     logger.info(f"Received search request: {search_req.query}")
6
7     # 1. Generate embedding for the query
8     try:
9         query_vector = await embedding_service.embed_text_async(search_req.
10 query)
11     except Exception as e:
12         raise EmbeddingError(f"Failed to embed query: {e}")
13
14     # 2. Search in Vector DB with optional filters
15     search_filters = search_req.filters or {}
16     if search_req.cluster_filter and search_req.cluster_filter != 'all':
17         search_filters['cluster'] = int(search_req.cluster_filter)
18
19     results = await vector_db.search(
20         query_vector=query_vector,
21         limit=search_req.limit,
22         filter_criteria=search_filters
23     )
24
25     # 3. Format results
26     formatted_results = []
27     for hit in results:
28         formatted_results.append(SearchResult(
29             id=str(hit.get("id")),
30             score=hit.get("score"),
31             text=hit.get("text"),
32             metadata=hit.get("metadata", {})
33         ))
34
35     return SearchResponse(results=formatted_results, count=len(
36 formatted_results))
```

Listing 7: Search endpoint from `main.py`.

5.2 Filtering Capabilities

The search system supports several filtering options:

- **Cluster Filtering:** Restrict results to a specific cluster (e.g., “Machine Learning” documents only).
- **Date Range:** Filter by upload date (implemented in Qdrant payload filter).
- **File Type:** Filter by original file extension (e.g., only PDFs).
- **Folder Path:** Search within a specific folder hierarchy.

Filters are applied at the Qdrant level using payload filtering, ensuring they don’t impact vector similarity computation.

5.3 Performance Optimization

Search performance is optimized through several mechanisms:

- **HNSW Index:** Qdrant uses Hierarchical Navigable Small World graphs for approximate nearest neighbor search, providing $O(\log n)$ query time.
- **Async Operations:** The entire search pipeline is asynchronous, preventing blocking during I/O-bound operations.
- **Caching:** The embedding service is a singleton, keeping the model loaded in memory.
- **Measured Performance:** Sub-50ms search latency for collections up to 10,000 vectors on consumer hardware.

6 Clustering and Organization

Automatic clustering organizes documents into semantically coherent groups without requiring manual categorization.

6.1 HDBSCAN Clustering

We use HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) for its ability to:

- Automatically determine the number of clusters
- Handle clusters of varying densities and arbitrary shapes
- Identify noise points (outliers) that don’t belong to any cluster
- Work well with high-dimensional embedding spaces

6.1.1 Algorithm Parameters

The clustering service is initialized with these parameters:

- **min_cluster_size:** Dynamically adjusted based on corpus size—3 for collections under 50 chunks, 5 for collections under 200 chunks, and 10 for larger collections. This ensures meaningful clusters are generated regardless of dataset size.
- **min_samples:** 3 (conservativeness of clustering)
- **metric:** euclidean (distance metric)

- **cluster_selection_method**: eom (Excess of Mass, better for variable density)

```

1 def fit_predict(self, embeddings: List[List[float]]) -> List[int]:
2     """Cluster embeddings and return cluster IDs (-1 = noise)."""
3     if not embeddings:
4         return []
5
6     if len(embeddings) < self.min_cluster_size:
7         # Not enough data for clustering
8         return [-1] * len(embeddings)
9
10    X = np.array(embeddings)
11
12    # Initialize HDBSCAN with EOM selection method
13    self.model = hdbscan.HDBSCAN(
14        min_cluster_size=self.min_cluster_size,
15        min_samples=self.min_samples,
16        metric=self.metric,
17        cluster_selection_method='eom'
18    )
19
20    self.labels_ = self.model.fit_predict(X)
21
22    # Calculate statistics
23    n_clusters = len(set(self.labels_)) - (1 if -1 in self.labels_ else 0)
24    n_noise = list(self.labels_).count(-1)
25
26    logger.info(f"HDBSCAN found {n_clusters} clusters and {n_noise} noise
27    points")
28    return self.labels_.tolist()

```

Listing 8: ClusteringService.fit_predict() from clustering.py.

6.2 Semantic Cluster Naming

After clustering, each cluster receives a descriptive name generated using TF-IDF keyword extraction.

6.2.1 Naming Algorithm

```

1 def generate_cluster_names(self, all_data: List[dict],
2                             cluster_labels: List[int]) -> dict:
3     """Generate semantic names for clusters using TF-IDF."""
4     from sklearn.feature_extraction.text import TfidfVectorizer
5
6     cluster_names = {}
7
8     # Group chunks by cluster
9     cluster_texts = {}
10    for item, label in zip(all_data, cluster_labels):
11        if label not in cluster_texts:
12            cluster_texts[label] = []
13            text = item.get('metadata', {}).get('text', '')
14            if text:
15                cluster_texts[label].append(text)
16
17    # Generate names for each cluster
18    for cluster_id, texts in cluster_texts.items():
19        if cluster_id == -1:
20            cluster_names[-1] = "Uncategorized"
21            continue
22
23        if len(texts) < 2:

```

```

24         cluster_names[cluster_id] = f"Cluster {cluster_id}"
25         continue
26
27     # Extract keywords with TF-IDF
28     n_docs = len(texts)
29     vectorizer = TfidfVectorizer(
30         max_features=5,
31         stop_words='english',
32         ngram_range=(1, 2), # 1-2 word phrases
33         min_df=2 if n_docs >= 5 else 1,
34         max_df=0.8 if n_docs >= 5 else 1.0
35     )
36
37     vectorizer.fit(texts)
38     keywords = vectorizer.get_feature_names_out()
39
40     # Take top 3, capitalize, join
41     top_keywords = [k.title() for k in keywords[:3]]
42     cluster_names[cluster_id] = " & ".join(top_keywords)
43
44     return cluster_names

```

Listing 9: ClusteringService.generate_cluster_names() from clustering.py.

6.2.2 Clustering Workflow

Figure 6 shows the complete clustering workflow.

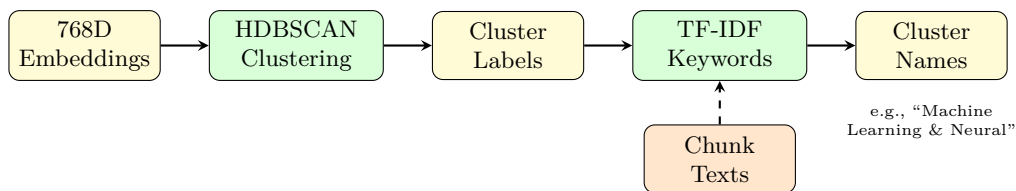


Figure 6: Clustering workflow from embeddings to semantic cluster names.

7 3D Visualization

The system includes an interactive 3D visualization of the embedding space, allowing users to explore semantic relationships between documents visually.

7.1 Dimensionality Reduction

To visualize 768-dimensional embeddings in 3D, we use Principal Component Analysis (PCA):

- **Input:** 768-dimensional embedding vectors
- **Output:** 3-dimensional coordinates for rendering
- **Properties:** Linear, deterministic, supports incremental projection

The projection is computed once and cached. New query embeddings can be projected using the stored transformation matrix without refitting.

7.1.1 Cold-Start Robustness

The `textttDimensionalityReducer` includes fallback logic for small datasets. When the number of

samples is less than the number of components (e.g., a “cold start” with only 1-2 documents), the system pads the output with zeros to prevent mathematical errors and system crashes. This ensures graceful degradation and reinforces the system’s production-readiness.

7.2 Interactive Visualization

The `embedding-visualizer.js` module (577 lines) implements the 3D visualization using Three.js:

7.2.1 Three.js Scene Setup

```
1 export class EmbeddingVisualizer {
2   constructor(container, options = {}) {
3     this.container = container;
4     this.baseUrl = options.baseUrl || 'http://127.0.0.1:8000';
5
6     // Scene setup
7     this.scene = new THREE.Scene();
8     this.scene.background = new THREE.Color(0x111111);
9
10    // Camera with perspective projection
11    this.camera = new THREE.PerspectiveCamera(
12      75, this.width / this.height, 0.1, 1000
13    );
14    this.camera.position.set(3, 3, 3);
15    this.camera.lookAt(0, 0, 0);
16
17    // WebGL renderer with antialiasing
18    this.renderer = new THREE.WebGLRenderer({ antialias: true });
19    this.renderer.setSize(this.width, this.height);
20    this.renderer.setPixelRatio(window.devicePixelRatio);
21
22    // Orbit controls for rotation, zoom, pan
23    this.controls = new OrbitControls(this.camera, this.renderer.domElement
24  );
25    this.controls.enableDamping = true;
26    this.controls.dampingFactor = 0.05;
27
28    // Lighting
29    const ambientLight = new THREE.AmbientLight(0xffffff, 0.6);
30    const directionalLight = new THREE.DirectionalLight(0xffffff, 0.8);
31    directionalLight.position.set(5, 10, 7);
32    this.scene.add(ambientLight, directionalLight);
33  }
```

Listing 10: EmbeddingVisualizer scene setup from `embedding-visualizer.js`.

7.2.2 Features

The visualization provides:

- **InstancedMesh:** Efficient rendering of thousands of points using GPU instancing.
- **Cluster Coloring:** Each cluster is assigned a unique color using HSL color space.
- **Query Highlighting:** When a search is performed, the query point appears as a gold sphere with lines connecting to the nearest neighbors.
- **Camera Guide Line:** A yellow line from the camera to the query point helps users locate search results.

- **Tooltips:** Hovering over points shows filename, ID, and cluster information.
- **Auto-scaling:** The camera automatically adjusts to fit all points in view.

7.2.3 Visualization Screenshot

Figure 7 shows the 3D visualization interface with clustered documents and a search query.

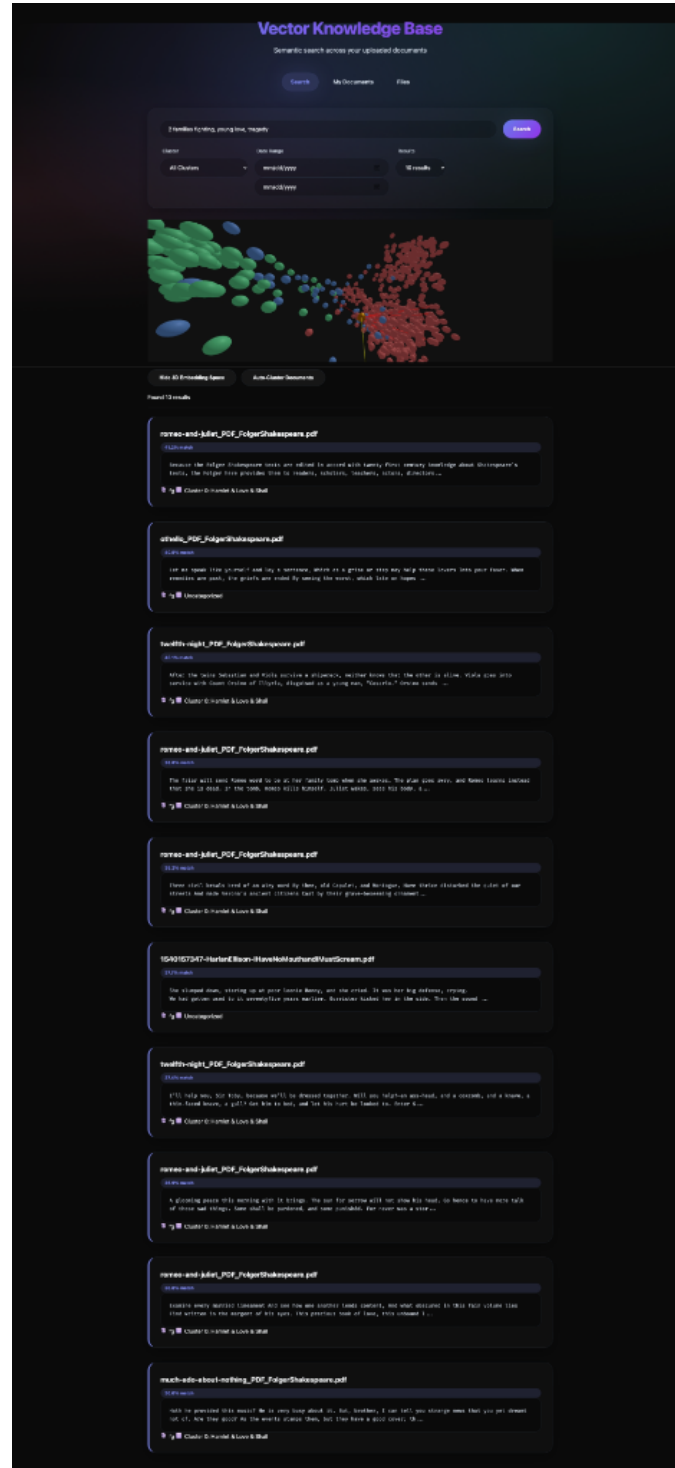


Figure 7: 3D embedding space visualization showing clustered documents (colored spheres) and a search query (gold sphere) with similarity lines to nearest neighbors.

8 File Management System

Beyond search, the system provides a full file organization layer with hierarchical folders and drag-and-drop management.

8.1 Folder Organization

8.1.1 SQLite Hierarchy

Folders are stored in SQLite with a recursive parent-child structure (see Section 3.4). Key features:

- **Arbitrary Nesting:** Folders can be nested to unlimited depth.
- **Root Level:** Files can be placed at the root (no folder) or remain “unsorted.”
- **Cascade Delete:** Deleting a folder moves its files to unsorted (not deleted).

8.1.2 Breadcrumb Navigation

The frontend displays breadcrumb paths for easy navigation:

Root > Schoolwork > Senior Year > Math

Users can click any breadcrumb segment to navigate directly to that level.

8.1.3 Drag-and-Drop

Files can be dragged between folders, between unsorted and folders, or to root. The frontend uses native HTML5 drag events, with the backend `/files/move` endpoint handling the database update.

8.2 Batch Upload

The batch upload feature preserves folder structure from the user’s machine.

8.2.1 Folder Structure Preservation

When uploading a folder, the system:

1. Reads the relative path of each file (e.g., `schoolwork/math/notes.pdf`)
2. Creates any missing folders in the hierarchy (using `get_or_create_folder_path`)
3. Assigns each file to its corresponding folder

8.2.2 Progress Tracking

Batch uploads display real-time progress:

- Current file count and total files
- Per-file processing status (extracting, embedding, storing)
- Overall progress percentage

8.2.3 File Organization Screenshot

Figure 8 shows the file organization interface.

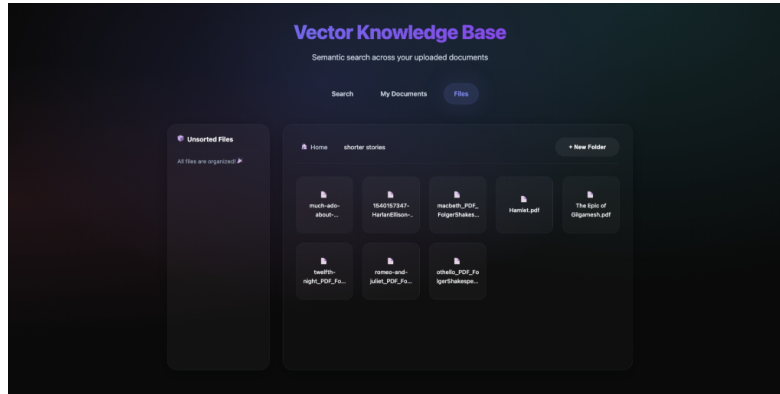


Figure 8: File organization interface showing folder hierarchy with breadcrumb navigation and file cards.

8.3 Data Export

The `/export` endpoint generates a ZIP archive that reconstructs the virtual folder hierarchy as a physical directory structure. The export logic:

1. Builds a complete path map from SQLite folder records (resolving parent chains)
2. Maps each physical file to its virtual folder location
3. Archives files into the ZIP with paths matching the user-defined hierarchy

This ensures data organization continuity when exporting data from the platform, preserving the semantic folder structure the user created rather than the flat physical storage.

9 API Design

The backend exposes a comprehensive REST API with 40+ endpoints, plus optional MCP integration for AI agents.

9.1 RESTful Endpoints

9.1.1 Core Endpoints

Table 4 lists the primary API endpoints.

Table 4: Core REST API endpoints.

Method	Endpoint	Description
GET	/health	Health check (returns “ok”)
POST	/upload	Upload single file with metadata
POST	/upload-batch	Batch upload files preserving folders
POST	/search	Semantic search with optional filters
GET	/documents	List all uploaded documents
DELETE	/documents/{filename}	Delete document and all chunks
DELETE	/reset	Clear all data (protected)
GET	/folders	List all folders
POST	/folders	Create new folder
PUT	/folders/{folder_id}	Rename or move folder
DELETE	/folders/{folder_id}	Delete folder
POST	/files/move	Move file to folder
POST	/api/cluster	Trigger clustering job
GET	/api/clusters	Get cluster information
GET	/api/embeddings/3d	Get 3D coordinates
POST	/api/embeddings/3d/query	Project query to 3D

9.1.2 Pydantic Models

Request and response schemas use Pydantic for validation:

```

1 class SearchRequest(BaseModel):
2     query: str
3     limit: int = 5
4     filters: Optional[Dict[str, Any]] = None
5     cluster_filter: Optional[str] = None
6
7 class SearchResult(BaseModel):
8     id: str
9     score: float
10    text: str
11    metadata: Dict[str, Any]
12
13 class SearchResponse(BaseModel):
14     results: List[SearchResult]
15     count: int

```

Listing 11: Pydantic request/response models.

9.2 Model Context Protocol (MCP)

The system integrates with AI agents via the Model Context Protocol, enabling tools like Claude Desktop to interact with the knowledge base.

9.2.1 MCP Architecture

Figure 9 shows the MCP integration architecture.

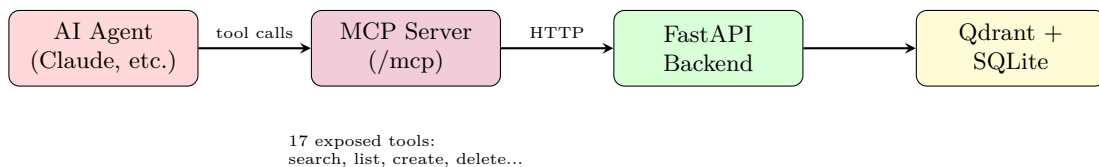


Figure 9: Model Context Protocol integration allowing AI agents to interact with the knowledge base.

9.2.2 MCP Tools

Table 5 lists the tools exposed to AI agents.

Table 5: MCP tools exposed to AI agents (18 included, 4 excluded).

Category	Tools
Health	health_check
Config	get_allowed_extensions
Search	search_documents
Documents	list_documents, delete_document, mcp_create_document
Folders	get_folders, create_folder, update_folder, delete_folder
Files	move_file, get_unsorted_files, get_files_in_folders
Clustering	cluster_documents, get_clusters
Visualization	get_embeddings_3d, transform_query_3d
Jobs	get_job_status
<i>Excluded</i>	<i>upload_file, upload_batch, reset_data, export_data</i>

The `mcp_create_document` tool allows AI agents to create text documents directly, bypassing file upload (useful for LLM-generated content).

9.2.3 Claude Desktop Integration

Figure 10 shows Claude Desktop using the MCP tools to interact with the knowledge base.

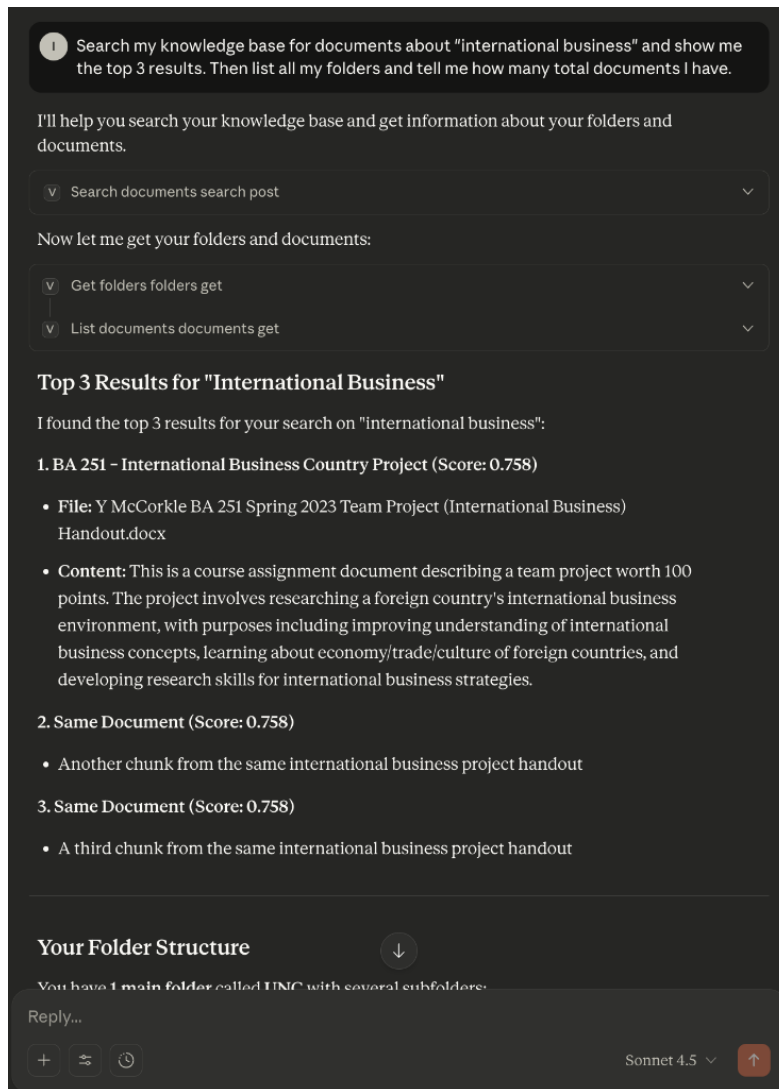


Figure 10: Claude Desktop using MCP tools to search and interact with the Vector Knowledge Base.

10 Deployment

The system supports two deployment modes: Docker (portable) and native (for GPU acceleration).

10.1 Docker Deployment

10.1.1 Docker Compose Configuration

The `docker-compose.yml` defines three services:

```

1 services:
2   qdrant:
3     image: qdrant/qdrant:latest
4     ports: ["6333:6333"]
5     volumes: [./qdrant_storage:/qdrant/storage]
6
7   backend:
8     build: .
9     ports: ["8000:8000"]

```

```

10     depends_on: [qdrant]
11     environment:
12         - QDRANT_HOST=qdrant
13         - MCP_ENABLED=true
14     volumes:
15         - ./uploads:/app/uploads
16         - ./backend_db:/app/data
17
18     frontend:
19         image: nginx:alpine
20         ports: ["8001:80"]
21         volumes: [./frontend:/usr/share/nginx/html:ro]

```

Listing 12: Docker Compose services.

10.1.2 Startup

A single command starts all services:

```
docker-compose up -d
```

The frontend is available at <http://localhost:8001>, backend API at <http://localhost:8000>.

10.2 Performance Mode (GPU)

For maximum performance, the backend can run natively with GPU acceleration.

10.2.1 Native Setup

1. Start Qdrant container only: `docker-compose up -d qdrant`
2. Create Python virtual environment
3. Install dependencies: `pip install -r requirements.txt`
4. Start backend: `uvicorn main:app -host 0.0.0.0 -port 8000`
5. Serve frontend: use any static file server on port 8001

10.2.2 Deployment Comparison

Table 6 compares deployment options.

Table 6: Deployment mode comparison (batch of 50 PDF pages).

Mode	Device	Time	Notes
Docker (default)	CPU	~18s	Portable, no GPU
Native (Apple)	MPS	~3s	Apple Silicon M1/M2/M3
Native (NVIDIA)	CUDA	~1s	High-end GPU

GPU acceleration provides 6-18x speedup for embedding generation, making native mode preferable for large batch uploads.

11 Security

While designed for personal or trusted-network use, the system includes security measures to prevent abuse.

11.1 Rate Limiting

11.1.1 SlowAPI Integration

Rate limiting is implemented using SlowAPI, a FastAPI-compatible rate limiter:

```
1 from slowapi import Limiter
2 from slowapi.util import get_remote_address
3
4 limiter = Limiter(key_func=get_remote_address)
5 app.state.limiter = limiter
6
7 @app.post("/upload")
8 @limiter.limit(settings.RATE_LIMIT_UPLOAD)
9 async def upload_file(request: Request, ...):
10     ...
```

Listing 13: Rate limiter configuration.

11.1.2 Configurable Limits

Default rate limits are set high for personal use but can be adjusted via environment variables:

- `RATE_LIMIT_UPLOAD`: 1000/minute (uploads)
- `RATE_LIMIT_SEARCH`: 1000/minute (searches)
- `RATE_LIMIT_RESET`: 60/minute (destructive operations)

11.2 Admin Protection

11.2.1 ADMIN_KEY Authentication

The `/reset` endpoint, which deletes all data, requires an admin key when configured:

```
1 @app.delete("/reset")
2 async def reset_data(admin_key: str = Header(None, alias="X-Admin-Key")):
3     if settings.ADMIN_KEY and admin_key != settings.ADMIN_KEY:
4         raise HTTPException(status_code=403, detail="Invalid admin key")
5     # Proceed with reset...
```

Listing 14: Admin key protection for reset.

The admin key is set via the `ADMIN_KEY` environment variable. If empty, protection is disabled (suitable for local development).

11.2.2 CORS Configuration

Cross-Origin Resource Sharing is restricted to specific origins:

```
CORS_ORIGINS=["http://localhost:8001", "http://127.0.0.1:8001"]
```

This prevents unauthorized web applications from accessing the API.

11.3 Input Sanitization

The ingestion pipeline includes robust filename sanitization to prevent security vulnerabilities and filesystem errors. The `sanitize_filename()` function in `ingestion.py` provides:

- **Path Traversal Prevention:** Strips directory components using `os.path.basename()`, preventing attackers from writing to arbitrary paths via filenames like `../../../../etc/passwd`.

- **Control Character Removal:** Removes null bytes (`\x00`) and other control characters that could cause unexpected behavior.
- **Windows-Reserved Characters:** Replaces characters illegal on Windows filesystems (`< > : " / \ | ? *`) with underscores, ensuring cross-platform compatibility.
- **Length Limiting:** Truncates filenames exceeding 200 characters while preserving the extension.

This input validation is critical for a system that interacts with file systems and prevents OS-level errors during file storage.

12 Performance Analysis

This section presents performance benchmarks and discusses scalability characteristics.

12.1 Benchmarks

12.1.1 Performance Metrics

Table 7 summarizes performance metrics measured on an Apple M1 Pro MacBook Pro.

Table 7: Performance benchmarks (M1 Pro, 16GB RAM, MPS acceleration).

Operation	Time	Notes
PDF extraction (10 pages)	1-2s	pypdf text extraction
Image OCR (single image)	2-5s	Tesseract processing
Embedding generation (per chunk)	50-100ms	With MPS acceleration
Embedding generation (per chunk)	300-500ms	CPU only (Docker)
Semantic search (10k vectors)	<50ms	Qdrant HNSW index
Semantic search (100k vectors)	100-200ms	Still sub-second
HDBSCAN clustering (1k points)	<1s	Including name generation
3D projection (PCA)	<100ms	For 1k embeddings
Full upload (50 PDFs)	3s	Native MPS mode
Full upload (50 PDFs)	18s	Docker CPU mode

12.1.2 GPU Acceleration Impact

GPU acceleration provides significant speedups:

- **MPS (Apple Silicon):** 6x faster than CPU
- **CUDA (NVIDIA):** 18x faster than CPU

The speedup is most noticeable during batch uploads where many embeddings are generated sequentially.

12.2 Scalability

12.2.1 Vector Storage Capacity

Qdrant can handle 100,000+ documents efficiently:

- HNSW index provides $O(\log n)$ search time
- Payload filtering doesn't significantly impact performance

- Disk-backed storage allows scaling beyond RAM

12.2.2 O(1) Document Listing

The JSON document registry provides O(1) document listing, avoiding expensive Qdrant scrolls:

```
# O(1) - read from JSON
documents = document_registry.list_all_documents()

# O(n) - avoided by using registry
documents = await vector_db.scroll_all()
```

12.2.3 3D Cache Strategy

The 3D visualization cache is invalidated when:

- New documents are uploaded
- Documents are deleted
- Clustering is re-run

This ensures the visualization reflects current data while avoiding redundant PCA computations.

13 Conclusion

13.1 Summary of Contributions

This paper presented the Vector Knowledge Base, a complete semantic document management system with the following contributions:

1. **Modular Extractor Architecture:** A factory pattern supporting 9 extractors across 23 file formats, including OCR for images and AST-aware processing for code.
2. **Intelligent Text Chunking:** Token-aware chunking with sentence boundary preservation for prose and AST-based chunking for Python code, ensuring semantic coherence of chunks.
3. **Automatic Clustering:** HDBSCAN clustering with TF-IDF-based semantic name generation, enabling automatic organization without manual categorization.
4. **3D Visualization:** PCA-based dimensionality reduction with Three.js rendering, providing intuitive exploration of the embedding space.
5. **AI Agent Integration:** Model Context Protocol (MCP) support enabling AI assistants like Claude Desktop to interact with the knowledge base programmatically.
6. **Production-Ready Architecture:** Docker deployment, GPU acceleration, rate limiting, and a complete REST API.

13.2 Future Work

Several directions for future development are identified:

- **Additional File Formats:** Support for additional formats such as ePub, audio transcription (Whisper), and video frame extraction.
- **Advanced Clustering:** Experimentation with alternative algorithms (OPTICS, spectral clustering) and hierarchical cluster visualization.

- **Clustering Write-Back Optimization:** The current implementation updates cluster metadata via individual `set_payload` calls per point— $O(n)$ network operations for n chunks. Batching these updates would significantly improve clustering speed for large corpora (100,000+ chunks).
- **Real-Time Collaboration:** Multi-user support with WebSocket-based real-time updates and shared collections.
- **Cross-Document Linking:** Automatic detection and visualization of semantic relationships between documents.
- **Fine-Tuned Embeddings:** Domain-specific embedding model fine-tuning for specialized use cases (legal, medical, technical).
- **Dependency Cleanup:** The `beautifulsoup4` library is included in requirements but currently unused; it may be removed or utilized for enhanced HTML parsing in future versions.

13.3 Final Remarks

The Vector Knowledge Base demonstrates that semantic search, intelligent clustering, and interactive visualization can be integrated into a cohesive, production-ready system. By leveraging modern tools—SentenceTransformers for embeddings, Qdrant for vector storage, HDBSCAN for clustering, and Three.js for visualization—the system provides capabilities previously limited to enterprise solutions while remaining accessible for personal knowledge management.

The open architecture, with clear separation between extraction, embedding, storage, and presentation layers, enables extension and customization for specific domains. The MCP integration further extends utility by enabling AI agents to leverage the knowledge base as a tool, bridging human and machine interaction with stored knowledge.

References

- [1] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3982–3992, Association for Computational Linguistics, 2019.
- [2] L. McInnes, J. Healy, and S. Astels, “hdbscan: Hierarchical density based clustering,” *The Journal of Open Source Software*, vol. 2, no. 11, p. 205, 2017.
- [3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, vol. 30, pp. 5998–6008, Curran Associates, Inc., 2017.
- [4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Association for Computational Linguistics, 2019.
- [5] Qdrant, “Qdrant vector database documentation.” <https://qdrant.tech/documentation/>, 2024. Accessed: 2025-12-01.
- [6] Various Authors, “Vector database comparison: Weaviate, milvus, and pinecone.” <https://weaviate.io/blog/vector-database-comparison>, 2024. Comprehensive comparison of vector database technologies including Weaviate, Milvus, Pinecone, and Qdrant.
- [7] Elastic, “Elasticsearch guide: Vector search.” <https://www.elastic.co/guide/en/elasticsearch/reference/current/dense-vector.html>, 2024. Traditional keyword search with vector capabilities, Accessed: 2025-12-01.
- [8] N. Reimers and I. Gurevych, “Sentencetransformers documentation.” <https://www.sbert.net/>, 2024. Accessed: 2025-12-01.
- [9] S. Ramírez, “Fastapi framework documentation.” <https://fastapi.tiangolo.com/>, 2024. Accessed: 2025-12-01.

A Configuration Reference

Table 8 lists all environment variables configurable via `.env`.

Table 8: Environment variables for system configuration.

Variable	Default	Description
<i>Qdrant Configuration</i>		
QDRANT_HOST	localhost	Qdrant server hostname
QDRANT_PORT	6333	Qdrant server port
QDRANT_COLLECTION	vector_db	Collection name
<i>File Upload</i>		
UPLOAD_DIR	uploads	Directory for uploaded files
MAX_FILE_SIZE	52428800	Max file size (50MB)
<i>Embedding Model</i>		
EMBEDDING_MODEL	all-mpnet-base-v2	SentenceTransformer model
DEVICE	auto	Compute device (auto/cpu/cuda/mps)
<i>Chunking</i>		
CHUNK_SIZE	500	Target chunk size (tokens)
CHUNK_OVERLAP	50	Overlap between chunks
<i>Security</i>		
ADMIN_KEY	(empty)	Key for /reset endpoint
RATE_LIMIT_UPLOAD	1000/minute	Upload rate limit
RATE_LIMIT_SEARCH	1000/minute	Search rate limit
RATE_LIMIT_RESET	60/minute	Reset rate limit
<i>MCP Server</i>		
MCP_ENABLED	true	Enable MCP server
MCP_PATH	/mcp	MCP endpoint path

B API Endpoint Quick Reference

Table 9 provides a complete API endpoint reference.

Table 9: Complete REST API endpoint reference.

Method	Endpoint	Description
<i>Core</i>		
GET	/health	Health check
GET	/config/allowed-extensions	List allowed file types
<i>Documents</i>		
POST	/upload	Upload single file
POST	/upload-batch	Batch upload with folder structure
GET	/documents	List all documents
DELETE	/documents/{filename}	Delete document by filename
DELETE	/reset	Clear all data (protected)
<i>Search</i>		
POST	/search	Semantic search with filters
<i>Folders</i>		
GET	/folders	List all folders
POST	/folders	Create folder
PUT	/folders/{folder_id}	Update folder
DELETE	/folders/{folder_id}	Delete folder
<i>Files</i>		
POST	/files/move	Move file to folder
GET	/files/unsorted	Get unsorted files
GET	/files/in_folders	Get files in folders
<i>Clustering & Visualization</i>		
POST	/api/cluster	Trigger clustering job
GET	/api/clusters	Get cluster information
GET	/api/embeddings/3d	Get 3D coordinates
POST	/api/embeddings/3d/query	Project query to 3D
<i>Jobs</i>		
GET	/api/jobs/{job_id}	Get job status
GET	/api/jobs	List all jobs
<i>MCP</i>		
POST	/mcp/create-document	Create document from text

C Supported File Types

Table 10 lists all 23 supported file extensions.

Table 10: Complete supported file type reference.

Category	Extensions	Extractor	Library
Documents	.pdf	PDFExtractor	pypdf
	.docx	DocxExtractor	docx2txt
	.pptx, .ppt	PptxExtractor	python-pptx
Images	.jpg, .jpeg, .png, .webp	ImageExtractor	pytesseract
Text	.txt, .md	TextExtractor	built-in
Code	.py, .js, .java, .cpp	CodeExtractor	built-in
	.html, .css, .json	CodeExtractor	built-in
	.xml, .yaml, .yml	CodeExtractor	built-in
	.cs	CsExtractor	custom regex
Data	.xlsx	XlsxExtractor	openpyxl
	.csv	CsvExtractor	built-in csv

D User Interface Screenshots

This appendix provides additional screenshots of the user interface.

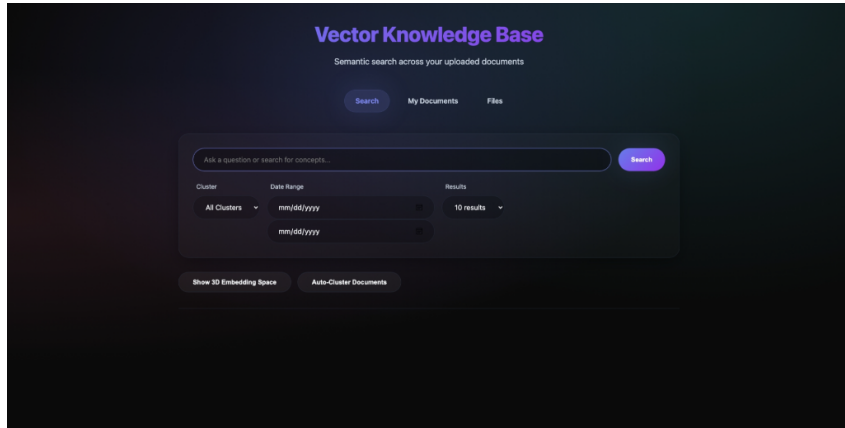


Figure 11: Main search interface with query input and cluster filter.

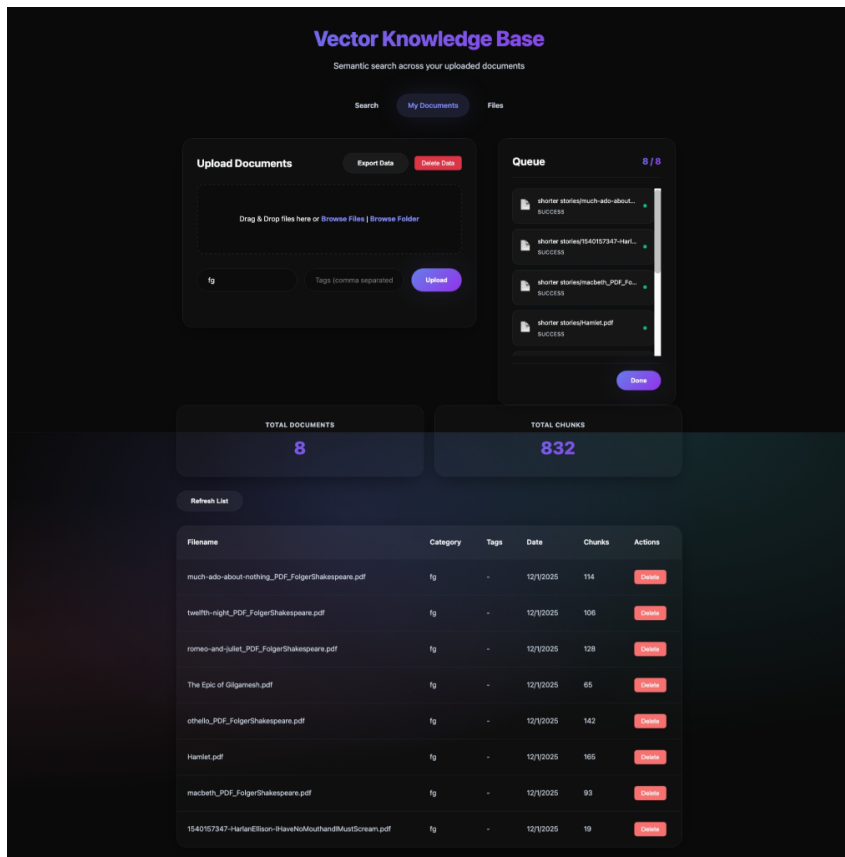


Figure 12: Documents page showing uploaded files with metadata and actions.

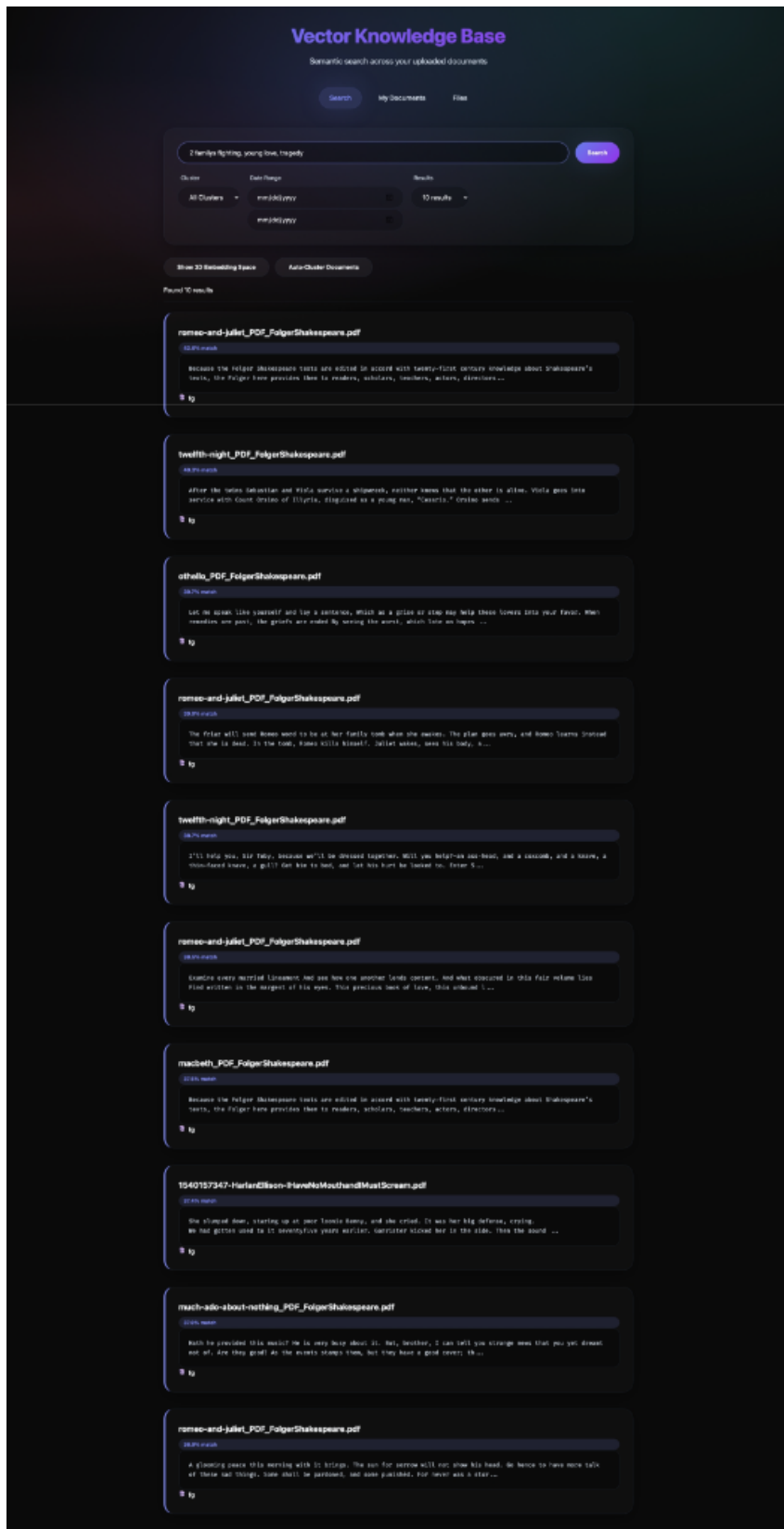


Figure 13: Search results displaying ranked document chunks with similarity scores.