Data Processing Pipeline

Nirmiti Patil

November 2025

1 Introduction

This project demonstrates a small, modular pipeline for extracting, cleaning, structuring, embedding, and storing unstructured textual data. The pipeline is split into clear components so each concern can be extended or replaced without affecting the rest of the system.

2 Architecture and Design Choices

2.1 Components

2.1.1 Loader

Handles file-type specific extraction (CSV, TXT, PDF, HTML). It returns a uniform list of items of the form `{source, text}`. The loader is directory-aware so it can process many files in a batch.

2.1.2 Cleaner

Applies deterministic, dependency-light normalizations (whitespace collapsing, removal of non-printable characters, snippet extraction). This keeps text predictable for downstream embedding and indexing.

2.1.3 EmbeddingModel

An abstraction over text embeddings. It prefers sentence-transformers for high-quality dense vectors, and falls back to a reproducible TF-IDF + SVD pipeline if that library isn't available. This provides a sensible accuracy vs. dependency tradeoff for demos and testing.

2.1.4 VectorStore

Pluggable storage for embeddings. If FAISS is present the implementation persists a FAISS index for efficient nearest-neighbor search. Otherwise it falls back to an in-memory matrix with persisted numpy/json artifacts and a brute-force cosine search. This ensures functionality even in restricted environments.

3 Why These Tools

3.1 SentenceTransformers

Balances performance and quality for semantic embeddings and is widely used in research and production.

3.2 FAISS

Provides a fast, memory-efficient nearest-neighbor backend when available. The code is written to still function when FAISS can't be installed, which simplifies testing and portability.

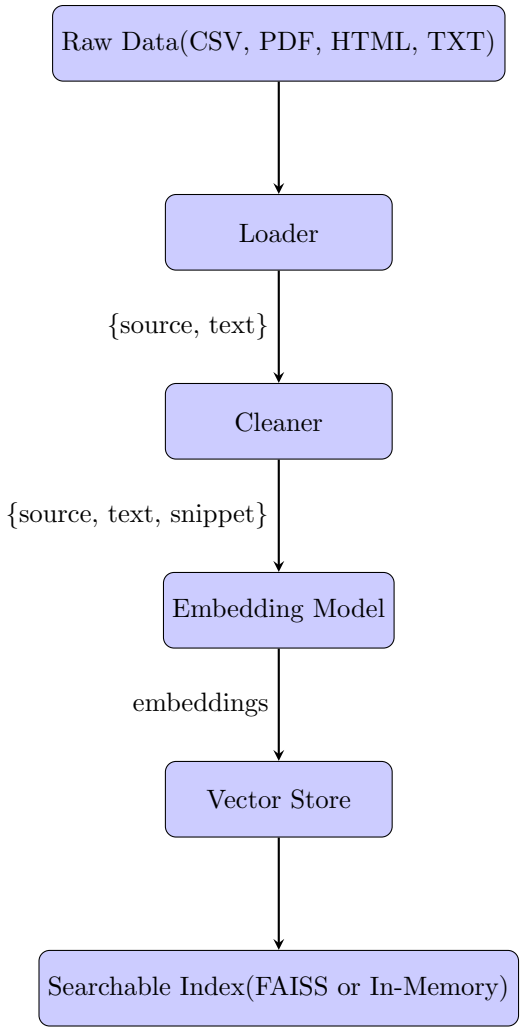

Figure 1: Data Processing Pipeline Architecture

4 Scalability Considerations

4.1 Modularity

Each component is a focused module, so you can scale or swap parts independently. For example, replace the Loader to add S3 or database sources, or swap embeddings with an external service.

4.2 Batch and Streaming

The loader design supports directory-level batching. For larger datasets you'd add chunked reads, async IO, and streaming writes to avoid holding all data in memory.

4.3 Vector Indexing

For production scale (millions of vectors) use an approximate nearest neighbor index (FAISS HNSW or IVF+PQ) stored on disk and sharded across machines. Also consider incremental indexing rather than rebuilding.

5 How to Run

5.1 Installation

Install the optional dependencies listed in `requirements.txt` when you want better embeddings and FAISS persistence. The code runs with minimal dependencies.

5.2 Execution

Run the pipeline with the following command to produce `data/cleaned_output.json` and a persisted index:

```
python main.py data/example_raw.csv
```