RAG News Retrieval System – Design & Usage Documentation

This document explains the end to end design of the RAG retrieval system built for the home assignment. It covers dataset choice, vector DB, embeddings, chunking strategy, retrieval parameters, edge cases, and how to use the application.

1. Dataset Selection & Preparation

1.1 Source dataset

Source: HuffPost News Category dataset (News_Category_Dataset_v3.json), a JSON Lines (JSONL) file where each line is a news article with: headline, short_description, category, link, authors, date. The json file is not provided in the GitHub repository since it was too big to upload, here is the link for the download: https://www.kaggle.com/datasets/rmisra/news-category-dataset?resource=download

1.2 Why this dataset

I chose this dataset because:

- It contains short, self contained snippets.
- Articles are labeled by category (POLITICS, TRAVEL, BUSINESS, etc.), which is useful metadata that helps with general search for news.
- It naturally supports user questions such as: “Show news about travel and hotels”, “Find articles about guns in U.S. politics”, “Stories related to health and vaccines”.

1.3 Subsetting & size constraints

The assignment requires unstructured text with a total text length under 30,000 characters. It also suggests ≤200 rows for small tabular datasets.

I satisfy this via project/data/jsonl_conversion_to_table.py:

- Load the JSONL file with pandas.
- Randomly sample 130 rows where short_description is non empty.
- Construct a text field as: "<headline>. <short_description> [Category: <category>, Date: <date>]".

- Save as project/data/news_rag_table.csv with columns: text (main content), category, date, link (metadata only).

I verified that there are 130 rows and a total of approx. 28,554 characters (below limit).

2. Vector Database Choice – Chroma

The assignment suggests Chroma, Pinecone, Weaviate, Mongo, pgvector, etc. I chose Chroma because it is local, lightweight, Python friendly, persistent on disk, and fits the scope of a small assignment.

Alternatives were not chosen because they add unnecessary complexity (Pinecone cloud service, Weaviate Docker based stack, Mongo/pgvector database setup).

3. Embeddings – Model

3.1 Model choice

Model: text-embedding-3-small (OpenAI). It is recommended in the assignment, low cost, low latency, and good for short news snippets.

4. Chunking Strategy

4.1 Options considered

I explored several chunking strategies while designing the system. My initial thought was that, since this is a tabular dataset where each row already represents a self contained piece of information, there's no real need to split rows into smaller chunks or to merge multiple rows into larger ones. I also considered grouping rows by article category (from the metadata), but quickly realized this would be counterproductive: assigning a single label to a large, mixed chunk based only on one category word would reduce generalization and could significantly degrade retrieval quality.

4.2 Final decision

I use one row = one chunk. Each row is already a coherent snippet, and doesn't need anything else to get a full context. Further splitting harms coherence and merging categories creates noisy oversized chunks.

4.3 Metadata vs chunk text

Only the text column is embedded. The metadata (category, date, link) is stored in Chroma but not embedded.

5. Vector Store Construction

Implemented in project/vector_store/build_chroma_store.py:

1. Load the CSV.
2. Initialize Chroma PersistentClient at project/vector_store/chroma_db.
3. Prepare texts, metadatas, and UUID ids.
4. Compute embeddings using OpenAI text-embedding-3-small.
5. Store documents, metadata, ids, and embeddings into the news_articles collection.

If the DB is corrupted, delete chroma_db/ and rebuild.

6. Retrieval Implementation

Implemented in project/rag/retriever.py.

Data structure: RetrievedChunk with text, category, date, link, score.

retrieve(query, top_k=4, similarity_threshold=0.25):

1. Validate input.
2. Detect single word vs multi word queries.
3. Adjust parameters:
    - Single word: top_k = 6, relaxed threshold.
    - Multi word: top_k = 3, normal threshold.

      This is done in order to allow for a more general search with random keywords and also for a specific search for an article.

4. Embed query and search.
5. Convert cosine distance to similarity as (1 - distance).

6. Apply base threshold and a global floor MIN_ACCEPTABLE_SCORE = -0.55 (so that most of the time only somewhat relevant articles are shown if nothing is really relevant).

7. Single word: return up to 6 exploratory results.

8. Multi word: return up to 3 focused results.

Design goals: avoid bad matches, support exploratory one word queries, and provide precise retrieval for multi word queries.

7. Minimal UI – Backend & Frontend

Backend: Flask (project/backend/app.py) with two routes:

- GET / → renders index.html with instructions.

- POST /search → processes query, runs retrieval, shows exploratory or focused messages, displays results.

Frontend: index.html with a centered input box, a search button, user messages, and a list of retrieved articles with rank, similarity score, category, date, text, and link.

8. How to Use the Application

8.1 Setup

Install dependencies:

```
cd /path/to/project
 pip install pandas chromadb openai flask
```

Set API key:

```
export OPENAI_API_KEY="sk-your-key-here"
```

8.2 Data & vector store

```
Optional: regenerate CSV (Make sure you downloaded the dataset from Kaggle)
 python3 data/jsonl_conversion_to_table.py

Build vector DB
 python3 vector_store/build_chroma_store.py
```

8.3 Run the UI

python3 backend/app.py
Visit http://localhost:8000.

Try single word queries (travel, politics, sports) for exploratory results, or multi word queries (travel hotels, politics guns, health vaccines) for focused results.

9. Summary of Key Design Choices

- Dataset: small subset of HuffPost news, 130 rows, under 30k characters.
- Chunking: one row per chunk.
- Vector DB: Chroma (local, persistent).
- Embeddings: text-embedding-3-small.
- Retrieval parameters: cosine → similarity = 1 – distance; threshold 0.25; top-k varies; global score floor −0.55.
- Backend: Flask.
- Frontend: minimal HTML/CSS.