Design and Implementation of an Intelligent Content Retrieval System

Buhle Mlandu

January 16, 2026

Abstract

This report presents the design, implementation, and evaluation of an Intelligent Content Retrieval System that enables semantic search over heterogeneous web-based content using modern natural language processing and vector database technologies. The system integrates ethical web scraping, structured text processing, transformer-based embedding generation, and similarity-based retrieval into a unified end-to-end pipeline. Content was collected from four publicly accessible sources, yielding over 28,000 words of text that were cleaned and segmented into 251 overlapping chunks to preserve contextual integrity.

Semantic embeddings were generated using the all-mpnet-base-v2 sentence transformer model and stored in a ChromaDB vector database configured for cosine similarity search. In addition to pure semantic retrieval, the system supports a hybrid search strategy that combines semantic similarity with keyword matching to improve retrieval precision. Retrieval quality was evaluated using manual relevance assessment, demonstrating consistent improvements in Precision@5 and mean relevance scores for hybrid retrieval.

The system further incorporates optional AI-assisted result enhancement using the Claude API to restructure retrieved text into coherent, query-focused responses without altering the original meaning. Overall, the results validate the effectiveness of hybrid semantic retrieval for intelligent information access.

Contents

1 Introduction 2
2 Website Selection and Data Collection 2
  2.1 Website Selection Rationale 2
  2.2 Selected Websites 3
  2.3 Ethical Data Collection Methodology 3
  2.4 Data Collection Summary 4
3 System Architecture and Design 4
  3.1 System Architecture Overview 4
  3.2 Data Flow Explanation 4
  3.3 Technology Stack 5
  3.4 Design Decisions and Trade-offs 5
4 Implementation Details 6
  4.1 Scraping Strategy and Challenges 6
  4.2 Text Processing Approach 7

4.3 Embedding Model Selection Rationale . . . 8
4.4 Vector Database Configuration . . . 9
4.5 Performance Optimisations . . . 10
5 Results and Analysis . . . 10
5.1 Statistics on Scraped Data . . . 10
5.2 Embedding Generation Metrics . . . 11
5.3 Query Performance Analysis . . . 11
5.4 Example Search Results with Interpretation . . . 11
5.5 Quality Assessment of Retrieval . . . 12
6 Challenges and Solutions . . . 15
6.1 Technical Obstacles Encountered . . . 15
6.2 Problem-Solving Approach . . . 16
6.3 Lessons Learned . . . 16
6.4 Future Improvements . . . 16

1 Introduction

This project focuses on the design and implementation of an intelligent content retrieval system that enables semantic search over web-based textual data. The assignment aims to demonstrate the practical application of modern information retrieval techniques, including ethical web scraping, text preprocessing, semantic embedding generation, and vector database-based retrieval.

The system extracts content from multiple publicly accessible websites across different domains, processes the text into structured chunks, and represents each chunk using transformer-based semantic embeddings. These embeddings are stored in a vector database to support similarity-based retrieval, allowing users to issue natural language queries and receive contextually relevant results. In addition to semantic retrieval, the system supports hybrid search strategies that combine semantic similarity with keyword filtering to improve precision.

To enhance usability, the system incorporates AI-assisted result restructuring, where retrieved text chunks are reorganized into clear, query-focused responses without adding new information or altering the original meaning. This report documents the system architecture, implementation approach, performance evaluation, and challenges encountered in building an end-to-end semantic content retrieval system in accordance with the assignment requirements.

2 Website Selection and Data Collection

2.1 Website Selection Rationale

The selection of websites for this project was guided by two primary criteria: compliance with the assignment requirement for content diversity and suitability for evaluating semantic search performance. To rigorously assess the system's ability to retrieve contextually relevant information, four websites were deliberately chosen from distinct and unrelated domains, minimizing topical overlap.

This strategic diversity enables clear evaluation of semantic retrieval accuracy. Queries related to machine learning, wildlife conservation, nuclear environmental impacts, or TensorFlow should retrieve content from the appropriate source without cross-domain confusion. The selected websites also differ significantly in writing style, structure, and terminology, allowing the system to be tested across encyclopedic reference material, narrative journalism, peer-reviewed scientific research, and technical documentation.

2.2 Selected Websites

Educational Resource – Wikipedia (Machine Learning)
URL: https://en.wikipedia.org/wiki/Machine_learning
Wikipedia was selected for its comprehensive and well-structured educational content. The machine learning article provides broad coverage of theoretical concepts, learning paradigms, algorithms, and applications. Its encyclopedic structure and balanced level of technical depth make it suitable for evaluating semantic retrieval across both introductory and advanced conceptual queries.

News Media – BBC Future (Dian Fossey and Gorilla Conservation)
URL: https://www.bbc.com/future/article/20251218-dian-fossey-the-woman-who-gave-her-life-to-save-the-
This source represents high-quality narrative journalism focused on wildlife conservation. The article employs a story-driven structure and domain-specific conservation vocabulary, providing a strong contrast to technical and academic sources. It enables evaluation of semantic retrieval from non-technical, human-centered content where key information is embedded within narrative context.

Research Publication – PubMed Central (Nuclear Testing Environmental Impact)
URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC4165831/
The selected peer-reviewed research article examines the long-term environmental and health impacts of nuclear weapons testing. The content is characterized by formal academic structure, dense scientific terminology, and evidence-based analysis. This source tests the system's ability to retrieve information from highly specialized research literature with minimal overlap with other domains.

Technical Documentation – TensorFlow (Introduction to Graphs)
URL: https://www.tensorflow.org/guide/intro_to_graphs
TensorFlow documentation was selected to represent developer-oriented technical material. The content explains computational graphs, execution models, and performance optimization concepts within a machine learning framework. This source supports evaluation of semantic retrieval for procedural and implementation-focused queries typical of software engineering and applied machine learning contexts.

2.3 Ethical Data Collection Methodology

All websites were scraped in accordance with ethical web scraping practices. Prior to data collection, each site's robots.txt file was checked to confirm permission for automated access.Where robots.txt could not be reliably fetched, manual verification was performed. Requests were rate-limited with randomized delays to minimize server load, and all HTTP requests included a descriptive User-Agent identifying the academic purpose of the project.

Only publicly accessible content was collected, and no authentication-restricted or paywalled material was accessed. Licensing and terms of use for all sources permit educational and research usage, ensuring legal and ethical compliance.

2.4 Data Collection Summary

The scraping process successfully collected content from all four selected websites, exceeding the assignment requirement of 5,000 characters per source. In total, over 28,000 words of textual content were gathered and preserved along with source metadata, including URLs, content category, and timestamps. The collected data provided a diverse and substantial corpus for subsequent text processing, embedding generation, and semantic retrieval evaluation.

3 System Architecture and Design

3.1 System Architecture Overview

The Intelligent Content Retrieval System is designed as a modular, end-to-end pipeline that transforms raw web content into semantically searchable knowledge. The system follows a linear processing flow in which each stage produces structured outputs that serve as inputs to the next stage, enabling clear separation of concerns and simplified evaluation.

Figure 1 illustrates the overall system architecture, highlighting the assignment core pipeline comprising web scraping, text processing, embedding generation, and vector database storage, as well as optional production extensions for API access, frontend interaction, and AI-assisted result enhancement.

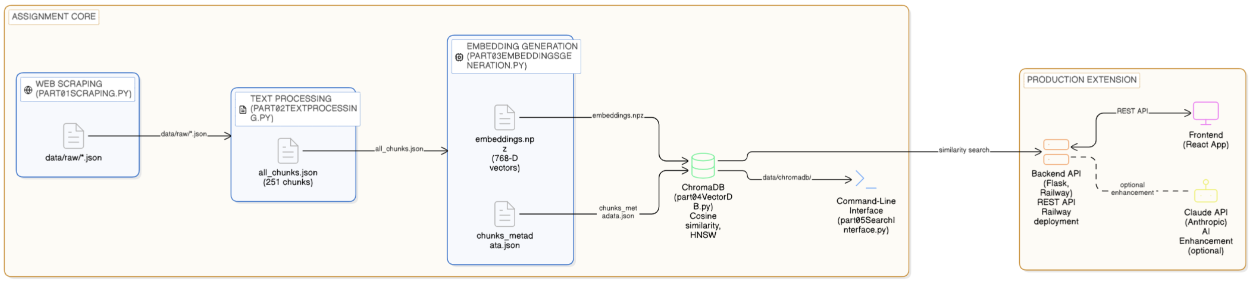

ASSIGNMENT CORE
WEB SCRAPING (PART01SCRAPING.PY)
data/raw/*.json
data/raw/*.json
TEXT PROCESSING (PART02TEXTPROCESSING.PY)
all_chunks.json (251 chunks)
all_chunks.json
EMBEDDING GENERATION (PART03EMBEDDINGSGENERATION.PY)
embeddings.npz (768-D vectors)
chunks_metadata.json
embeddings.npz
chunks_metadata.json
ChromaDB (part04VectorDB.py) Cosine similarity, HNSW
data/chromadb/
Command-Line Interface (part05SearchInterface.py)
similarity search
PRODUCTION EXTENSION
Backend API (Flask, Railway) REST API Railway deployment
REST API
Frontend (React App)
optional enhancement
Claude API (Anthropic) AI Enhancement (optional)

Figure 1: System Architecture Diagram Showing Assignment Core and Production Extensions

3.2 Data Flow Explanation

The system operates through a staged data flow. Web scraping modules collect textual content from publicly accessible websites and store the data in structured JSON format. The text processing stage cleans and segments the content into overlapping chunks to preserve contextual continuity. Each chunk is then converted into a high-dimensional semantic embedding using a transformer-based sentence embedding model.

Embeddings and associated metadata are stored in a persistent ChromaDB vector database configured for cosine similarity search. At query time, user input is embedded using the same model and compared against stored vectors to retrieve the most contextually relevant content. Retrieved results are accessed through a command-line interface or REST API, with optional AI-assisted restructuring applied to improve response clarity without modifying the original content.

3.3 Technology Stack

The system is implemented primarily in Python and leverages modern libraries for web scraping, natural language processing, and vector-based retrieval. Core components include `requests` and `BeautifulSoup4` for data collection, `ftfy` and `clean-text` for text preprocessing, `sentence-transformers` with the *all-mpnet-base-v2* model for embedding generation, and `ChromaDB` for persistent vector storage and similarity search. Optional production extensions are implemented using `Flask` for API services and the `Anthropic` SDK for AI-based result enhancement.

All library dependencies and exact version numbers used in the implementation are documented in the accompanying `requirements.txt` file to ensure reproducibility and environment consistency.

3.4 Design Decisions and Trade-offs

A modular pipeline architecture was adopted to improve maintainability and allow individual stages of the system to be executed and evaluated independently. This design simplifies debugging and avoids repeating expensive operations such as web scraping and embedding generation.

Transformer-based semantic embeddings were selected over keyword-based retrieval to enable context-aware search across heterogeneous content. The *all-mpnet-base-v2* model was chosen as it offers a strong balance between embedding quality and computational efficiency, producing 768-dimensional vectors with high semantic similarity performance while remaining suitable for interactive query workloads on the 251-chunk corpus. Smaller models such as *all-MiniLM-L6-v2* provided faster embedding generation but lower retrieval quality, while larger models introduced unnecessary computational overhead without proportionate gains for the dataset scale used in this project.

ChromaDB was selected as the vector database because it is lightweight, beginner- friendly, and recommended for academic use. It provides a Python-native API, persistent local storage, and built-in metadata support, making it more suitable for this project than more complex or cloud-based alternatives.

Text chunking with overlap was employed to preserve semantic context across chunk boundaries, improving retrieval quality at the cost of modest additional storage. To optimize query performance, a global cache was introduced to store previously retrieved results, reducing redundant vector database queries for repeated requests. Finally, AI-assisted result enhancement was implemented as an optional layer to improve response clarity without altering the original content.

4 Implementation Details

This section describes the technical implementation of the end-to-end content retrieval system, covering web scraping, text processing, embedding generation, vector database configuration, and key performance optimizations.

4.1 Scraping Strategy and Challenges

The web scraping component (`part01Scraping.py`) was implemented as a structured pipeline designed to prioritise ethical data collection, robustness, and reproducibility. For each source, the scraper executes seven sequential steps.

4.1.1 Implementation Strategy

Robots.txt compliance. Prior to any HTTP request, the system checks scraping permissions by parsing `robots.txt` using `urllib.robotparser.RobotFileParser`. The target domain and path are extracted from the URL to construct the corresponding `robots.txt` location, after which the `can_scrape()` routine determines whether the configured User-Agent is allowed to access the requested path. If `robots.txt` cannot be read, the implementation defaults to allowing access unless explicitly denied.

Request execution and error handling. Content is retrieved using the `requests` library with a 15-second timeout to prevent indefinite blocking on slow connections. Requests include a descriptive User-Agent string with academic context and contact information, alongside appropriate `Accept` and `Accept-Language` headers. HTTP errors are handled centrally via `response.raise_for_status()`, enabling consistent failure reporting and controlled recovery.

HTML parsing. Returned HTML is parsed using BeautifulSoup4 with the `html.parser` backend, providing reliable performance and tolerance of imperfect markup. This parsed representation supports targeted extraction via structural selectors.

Category-specific content extraction. Because site layouts differ substantially by content type, the scraper applies category-aware selectors to locate primary content containers. For example, Wikipedia extraction targets the article body, BBC extraction targets the semantic `<article>` element, TensorFlow documentation extraction targets the main documentation `<article>` container, and PubMed extraction targets the page's main content region. If the expected container is missing, the scraper logs the issue (potentially indicating a site structure change) and skips the source to avoid storing non-content noise.

Structure-preserving text extraction. A custom `get_smart_text()` routine converts HTML to clean text while preserving document structure. The function distinguishes block-level elements (e.g., paragraphs, headings, list items) that introduce line breaks from inline elements that should be separated by spaces. Non-content elements such as `<script>`, `<style>`, `<noscript>`, `<svg>`, and `<iframe>` are excluded entirely. Post-processing collapses redundant whitespace and normalises newlines, yielding readable text with retained paragraph boundaries.

Content validation. Extracted content is validated using character and word counts. A minimum threshold of 5,000 characters is used to detect insufficient sources (warning-only to support auditability). In the final dataset, all sources exceeded the threshold, with character counts ranging from approximately 19,000 to 120,000.

Structured persistence. Each scraped output is stored as JSON, containing the raw text and metadata fields such as URL, domain, category, timestamp, character count, word count, status code, and content type. Serialisation uses `ensure_ascii=False` to preserve Unicode and `indent=2` for readability, and files are written under `data/raw/` by category.

Rate limiting. To reduce server load and avoid detectable request patterns, a random delay of 2–4 seconds is applied between requests using `time.sleep(random.uniform(2, 4))`.

4.1.2 Challenges Encountered

Several practical challenges were addressed during development:

Wikipedia access constraints. The initial `robots.txt` check blocked Wikipedia requests for the configured User-Agent. The final implementation included a conditional override for Wikipedia URLs under an educational-use rationale, while retaining strict `robots.txt` compliance for all other sources (robot.txt for Wikipedia page was checked manually).

Heterogeneous HTML structures. Each site employed different conventions for marking primary content, requiring manual inspection and category-specific selectors. A dispatch-style strategy was implemented to apply the appropriate selector per category, with explicit fallback handling.

Loss of paragraph boundaries in naive extraction. BeautifulSoup's default `.get_text()` produced flattened output and sometimes included non-content text. The custom structure-aware extraction resolved both issues.

Encoding artefacts and special characters. Some sources exhibited encoding corruption. This was addressed via Unicode-preserving JSON output and downstream correction using `ftfy` during text processing.

4.2 Text Processing Approach

The text processing module (`part02TextProcessing.py`) transforms raw scraped documents into clean, semantically meaningful chunks suitable for embedding and retrieval. The pipeline consists of four stages: loading, cleaning, chunking, and writing with metadata enrichment.

4.2.1 Stage 1: Loading

All raw JSON files are discovered using `glob.glob`, enabling flexible handling of source filenames. Each file is loaded with UTF-8 decoding and validated for required fields (e.g., `content`, `category`, and `metadata`). Invalid or corrupt files are logged and skipped to avoid pipeline failure, while missing-input conditions raise a clear error instructing execution of the scraping stage.

4.2.2 Stage 2: Cleaning

Cleaning is performed using three layers: (i) encoding correction with `ftfy.fix_text`, (ii) standardised removal of noise using `cleantext.clean` (e.g., URLs, emails, phone numbers, currency symbols, emoji), and (iii) regular-expression normalisation of whitespace and line endings. Capitalisation and punctuation are preserved to retain semantic cues and sentence structure for transformer-based embedding models.

4.2.3 Stage 3: Intelligent Chunking

Chunks are generated using a boundary-aware strategy to balance semantic coherence and retrieval granularity. The implementation targets approximately 1,000 characters per chunk (within the 800–1,200 requirement) with a 150-character overlap. Where possible, chunk boundaries are aligned to sentence endings within a bounded search window; if no sentence boundary is found, the algorithm falls back to a word boundary to prevent splitting tokens.

4.2.4 Stage 4: Metadata Enrichment and Writing

Each chunk is assigned a unique identifier of the form `{category}_chunk_{index}` and is stored with enriched metadata, including source URL, category, domain, chunk index, per-source chunk count, character and word counts, and the original scraping timestamp. All chunks are written to a consolidated file (`data/processed/all_chunks.json`) to simplify downstream embedding and database ingestion.

4.2.5 Processing Statistics

The processing pipeline produced 251 total chunks. Category-level distribution was: Educational (128), News (20), Research Publication (69), and Technical Documentation (34). Across all chunks, the total character count was 273,800 and the total word count was 42,747. The average chunk length was 1090 characters.

4.3 Embedding Model Selection Rationale

The system uses `all-mpnet-base-v2` for embedding generation due to its favourable balance between retrieval quality and computational efficiency. The model produces 768-dimensional sentence embeddings that capture rich semantic structure while remaining practical for interactive query latency on a small-to-medium corpus.

4.3.1 Technical Implementation

The embedding module (`part03EmbeddingsGeneration.py`) encodes chunk text in batches to improve throughput. Embeddings are generated using `model.encode` with `batch_size=32` and `normalize_embeddings=True`. Normalisation ensures each embedding has unit L2 norm, enabling cosine similarity to be computed efficiently. Normalisation correctness is verified by computing vector norms and validating closeness to 1.0 within a small tolerance.

Embeddings are stored using `np.savez_compressed` to reduce disk usage while retaining associated metadata (e.g., model name, dimensionality, and chunk count). Chunk metadata is stored separately in JSON to support independent inspection and reuse.

4.3.2 Performance Metrics

Embedding generation performance was measured directly in the implementation using wall-clock timestamps recorded immediately before and after the embedding computation. For the final corpus of 251 text chunks, embedding generation completed in 274.48 seconds, corresponding to an average processing time of approximately 1.09 seconds per chunk and a throughput of 0.9 chunks per second. All generated embeddings were verified to be unit-normalised by computing their L2 norms and confirming closeness to 1.0 within a small numerical tolerance, ensuring compatibility with cosine similarity–based retrieval. No explicit instrumentation of CPU or memory utilisation was performed during embedding generation.

4.4 Vector Database Configuration

4.4.1 ChromaDB Setup

The vector database layer (`part04VectorDB.py`) uses ChromaDB with persistent storage via `chromadb.PersistentClient(path="data/chromadb")`, ensuring that indexing and stored embeddings survive across sessions. A collection is created (or loaded) with cosine distance configured as the similarity metric. Collection metadata records key configuration details such as model name and embedding dimensionality, supporting traceability and reproducibility.

4.4.2 Distance Metric and Indexing

Cosine similarity is well-suited to unit-normalised transformer embeddings because semantic meaning is primarily encoded in vector direction rather than magnitude. ChromaDB implements approximate nearest neighbour search via an HNSW index, which provides efficient retrieval with logarithmic-like search behaviour in practice. For the dataset scale used here (251 documents), this indexing strategy provides fast query response while maintaining high recall.

4.4.3 Batch Insertion and Persistence Verification

To reduce API overhead and accelerate ingestion, documents are inserted in batches of 100. Each batch supplies aligned arrays of IDs, embeddings, document texts, and metadata. Persistence is verified by re-instantiating the client and confirming the collection count matches the expected 251 items.

4.4.4 Query Execution and Optional Filtering

Queries are embedded using the same model and normalisation settings to preserve vector-space compatibility. Retrieval uses `collection.query` with `n_results` controlling top-k output. Optional metadata filtering is enabled via the `where` clause (e.g., filtering by `source_category`), supporting category-constrained retrieval without separate collections.

4.4.5 Hybrid Search and Re-ranking

In addition to dense semantic retrieval, the search interface implements a hybrid retrieval mode that combines cosine-based semantic similarity with lightweight keyword matching. The system first over-retrieves a candidate set using semantic search (up to $10k$, capped at 100 results) and then computes a keyword coverage score representing the proportion of user-provided keywords present in each candidate document. A combined hybrid score is computed as a weighted sum of the semantic similarity score and the keyword score, after which candidates are re-ranked by the hybrid score and the top-k results are returned. In the implementation, the default weighting prioritises semantic similarity (0.7) while still rewarding keyword matches (0.3); these weights are configurable via the interactive interface.

4.5 Performance Optimisations

Several optimisations were applied to reduce latency and improve usability:
Unit-normalised embeddings. Normalised vectors allow cosine similarity to be computed as a dot product, avoiding repeated norm computations during retrieval.

Batched embedding generation. Encoding chunks in batches reduces per-call overhead and improves hardware utilisation, yielding a substantial throughput improvement over per-chunk encoding.

Batched database insertion. Ingesting documents in batches reduces repeated index update overhead and minimises API-call costs during database population.

Persistent storage. Persisting the ChromaDB index eliminates database rebuild time across sessions, reducing repeated setup overhead during development and deployment.

Compressed embedding storage. Using compressed NumPy storage reduces disk footprint with negligible decompression overhead at this corpus scale.

Collectively, these optimisations are intended to reduce computational overhead during embedding generation and retrieval, supporting interactive query behaviour.

5 Results and Analysis

5.1 Statistics on Scraped Data

Four publicly accessible English sources were scraped across four categories (Educational, News, Research Publication, Technical Documentation). Table 1 summarizes the final cleaned text volume per source. After pre-processing and chunking (800–1200 characters with overlap), the processed corpus contained **251 chunks**, totalling **273,800 characters** and **42,747 words**.

Table 1: Scraped content statistics per source (post-cleaning).

Category	Domain	Characters	Words
Educational	en.wikipedia.org	123,245	18,595
News	www.bbc.com	19,007	3,227
Research Publication	pmc.ncbi.nlm.nih.gov	66,440	10,006
Technical Documentation	www.tensorflow.org	35,232	5,201
Total		**243,924**	**37,029**

Table 2: Chunk-level corpus summary.

Metric	**Value**
Total chunks	251
Total characters (all chunks)	273,800
Total words (all chunks)	42,747
Chunks by category	Educational: 128; News: 20; Research: 69; Tech Docs: 34

5.2 Embedding Generation Metrics

Semantic embeddings were generated using `all-mpnet-base-v2` (768 dimensions). Vectors were L2-normalised to support cosine similarity. Embeddings were generated in batches of 32 and stored for vector search.

Table 3: Embedding generation summary.

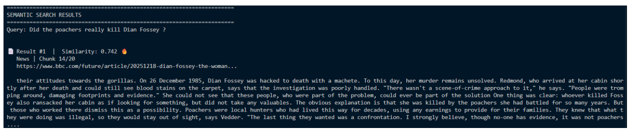
Metric	**Value**
Model	`all-mpnet-base-v2`
Embedding dimension	768
Total embeddings (chunks)	251
Batch size	32
Generation time	274.48 seconds
Normalised vectors	Yes
Embedding file size	0.74 MB

5.3 Query Performance Analysis

Two retrieval modes were evaluated: (i) **Semantic search** using cosine similarity over embeddings, and (ii) **Hybrid search** combining semantic similarity with keyword matching. Hybrid scoring improved overall retrieval quality (Section 5.5), showing higher Precision@5 and higher mean relevance ratings on a manual evaluation set (14 queries with the top-5 results evaluated for both semantic and hybrid search, resulting in a total of 140 manually assessed results.).

5.4 Example Search Results with Interpretation

Figure 2 illustrates results for the query: *"Did the poachers really kill Dian Fossey?"* Semantic search returned a BBC News chunk discussing Fossey's death and the uncertainty around the perpetrator, yielding a similarity score of **0.742**. Hybrid search produced the same highly relevant source but increased the combined score to **0.819** due to strong keyword overlap (keyword score **1.000**), demonstrating the benefit of lexical signals for named entities and explicit terms.

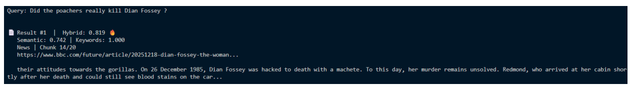


```
======================================================================
SEMANTIC SEARCH RESULTS
======================================================================
Query: Did the poachers really kill Dian Fossey ?


📄 Result #1  |  Similarity: 0.742 🔥
   News | Chunk 14/20
   https://www.bbc.com/future/article/20251218-dian-fossey-the-woman...

   their attitudes towards the gorillas. On 26 December 1985, Dian Fossey was hacked to death with a machete. To this day, her murder remains unsolved. Redmond, who arrived at her cabin shor
tly after her death and could still see blood stains on the carpet, says that the investigation was poorly handled. "There wasn't a scene-of-crime approach to it," he says. "People were trom
ping around, damaging footprints and evidence." She could not see that these people, who were part of the problem, could ever be part of the solution One thing was clear: whoever killed Foss
ey also ransacked her cabin as if looking for something, but did not take any valuables. The obvious explanation is that she was killed by the poachers she had battled for so many years. But
 those who worked there dismiss this as a possibility. Poachers were local hunters who had lived this way for decades, using any earnings to provide for their families. They knew that what t
hey were doing was illegal, so they would stay out of sight, says Vedder. "The last thing they wanted was a confrontation. I strongly believe, though no-one has evidence, it was not poachers
....
```

(a) Semantic retrieval (score 0.742).



```
Query: Did the poachers really kill Dian Fossey ?


📄 Result #1  |  Hybrid: 0.819 🔥
   Semantic: 0.742 | Keywords: 1.000
   News | Chunk 14/20
   https://www.bbc.com/future/article/20251218-dian-fossey-the-woman...

   their attitudes towards the gorillas. On 26 December 1985, Dian Fossey was hacked to death with a machete. To this day, her murder remains unsolved. Redmond, who arrived at her cabin shor
tly after her death and could still see blood stains on the car...
```

(b) Hybrid retrieval (combined 0.819).

Figure 2: Example query outputs showing semantic vs hybrid scoring behaviour.

5.5 Quality Assessment of Retrieval

Retrieval quality was assessed using **manual relevance evaluation** over **14 diverse queries**. For each query, the top-5 results were rated on a 3-point scale: 0 (Not Relevant), 1 (Somewhat Relevant), 2 (Highly Relevant), producing **70 results** per method.

Hybrid search improved performance consistently: Precision@5 increased from **84.3%** to **85.7%** (+1.4%), mean rating increased from **1.30** to **1.47** (+0.17), and the proportion of **Highly Relevant** results increased from **45.7%** to **61.4%**. This indicates that hybrid retrieval not only maintains strong top-k precision, but also shifts more results into the highest relevance category.

Table 4: Manual relevance evaluation summary (14 queries, top-5 results each).

Metric	**Semantic**	**Hybrid**
Precision@5	84.3%	85.7%
Mean rating (0–2)	1.30	1.47
Highly relevant (2)	45.7%	61.4%
Somewhat relevant (1)	38.6%	24.3%
Not relevant (0)	15.7%	14.3%

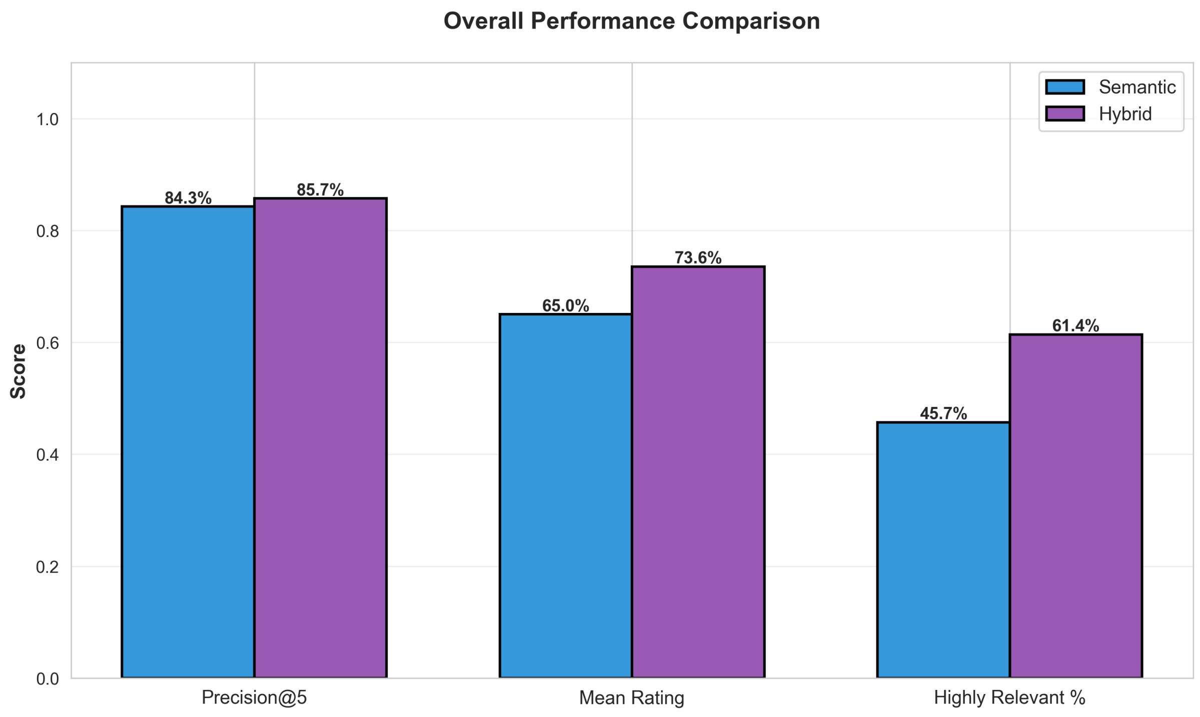


Figure 3: Overall comparison of Precision@5, mean rating, and Highly Relevant percentage.

Figure 3 compares overall retrieval performance between semantic and hybrid search across three aggregate metrics: Precision@5, mean relevance rating, and the proportion of highly relevant results. Precision@5 was computed by treating results rated as Somewhat Relevant (1) or Highly Relevant (2) as relevant. Mean rating represents the average graded relevance score across all evaluated results. The results show that hybrid search consistently outperforms pure semantic retrieval across all metrics, with the largest improvement observed in the proportion of highly relevant results, indicating improved ranking quality rather than marginal gains in recall.

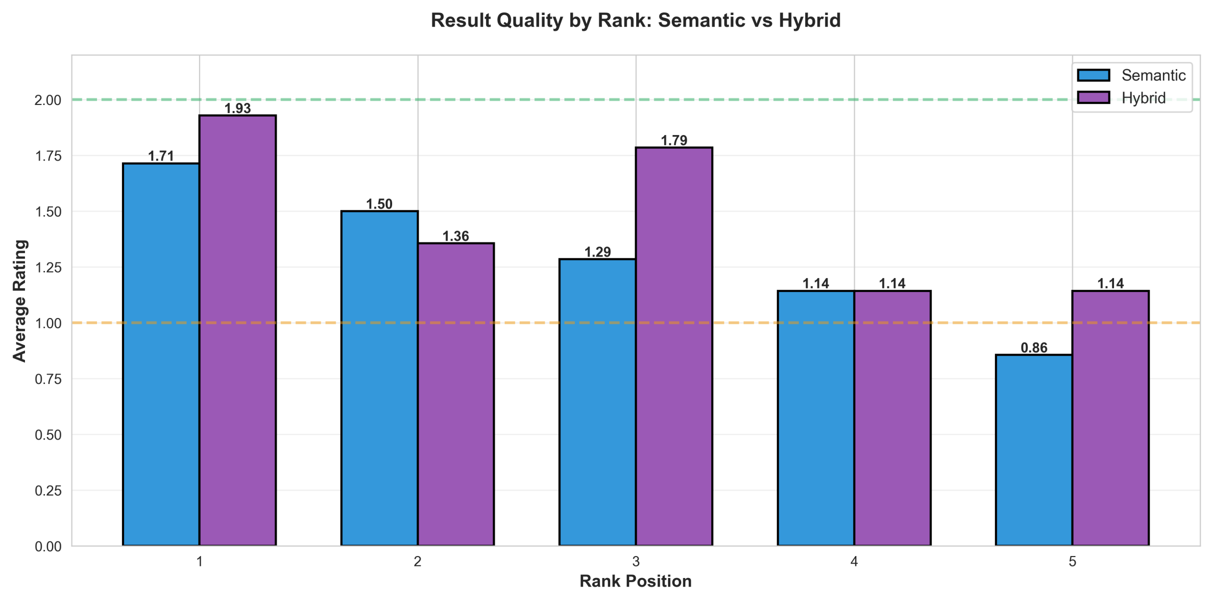


Figure 4: Average relevance by rank position (1–5). Hybrid improves early ranks and maintains stronger relevance deeper into the list.

Figure 4 illustrates the average relevance rating at each rank position (1–5) across all evaluated queries. Ratings were averaged per rank to assess how result quality degrades deeper into the ranked list. Hybrid search achieves higher average relevance at the top-ranked positions, particularly at ranks 1 and 3, while maintaining comparable or improved relevance at lower ranks. This demonstrates that hybrid scoring improves early precision, which is especially important for user-facing search scenarios where only the top few results are typically inspected.

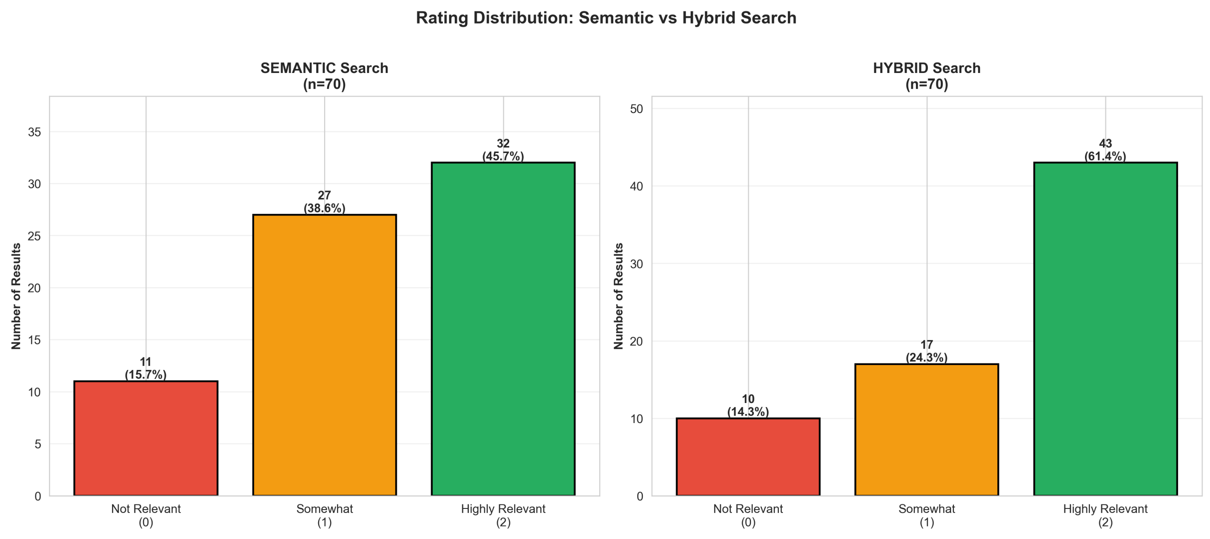


Figure 5: Distribution of relevance ratings for semantic vs hybrid search (n=70 results each).

Figure 5 presents the distribution of manual relevance ratings for semantic and hybrid search across all evaluated results ($n = 70$ per method). Each bar represents the total number of results assigned to each relevance category. Compared to semantic retrieval, hybrid search produces a substantially higher proportion of highly relevant results while reducing the number of marginally relevant outcomes. This shift in the distribution confirms that hybrid retrieval improves result quality in a qualitative sense, not merely by small numerical improvements in aggregate scores.

6 Challenges and Solutions

6.1 Technical Obstacles Encountered

Web Scraping and Content Extraction. Two primary challenges arose during the scraping phase. First, the robots.txt compliance check failed for Wikipedia due to HTTP header issues when attempting to automatically fetch and parse the robots.txt file. As a result, `RobotFileParser` was unable to retrieve the file and incorrectly blocked scraping, despite the target page (`/wiki/Machine_learning`) being explicitly permitted. Manual inspection of Wikipedia's robots.txt confirmed that the page was allowed. To handle this edge case, the implementation was modified to gracefully degrade by assuming scraping is permitted when robots.txt cannot be fetched, while continuing to strictly enforce robots.txt compliance for all other websites. In addition, the URL handling logic was corrected to pass only the path component (rather than the full URL) to `can_fetch()`, preventing false negatives on compliant pages.

Second, naive HTML text extraction using generic methods produced noisy outputs where navigation menus, advertisements, and script content were interleaved with core textual content. To address this, a custom context-aware extraction function was implemented. Block-level HTML elements triggered paragraph breaks, inline elements preserved appropriate spacing, and non-content tags such as `script` and `style` were excluded entirely. This significantly improved text coherence and downstream retrieval quality.

Memory Constraints in Cloud Deployment. When deploying the Flask-based API to Railway, the application encountered memory limitations during startup due to simultaneous loading of the sentence transformer model and the full ChromaDB collection. An initial attempt to mitigate this through lazy loading proved unreliable under production conditions. As a result, the issue was resolved by upgrading to a higher-memory Railway service tier. This ensured stable application startup while preserving deterministic behavior and avoiding runtime loading complexity.

Embedding Dimension Mismatch. Early ChromaDB queries failed due to embedding dimension mismatches. While embeddings were correctly generated using a 768-dimensional transformer model, the database was initially configured to use a default embedding function. This was corrected by explicitly supplying pre-computed embeddings during collection creation and enforcing consistent model usage at query time. Additional validation checks were introduced to ensure dimensional consistency before executing search operations.

Hybrid Search Result Limitations. The initial hybrid retrieval strategy used a limited candidate pool prior to keyword scoring, occasionally yielding fewer than the desired number of results. This was resolved by substantially increasing the semantic candidate pool before hybrid scoring and replacing hard keyword filtering with weighted score integration. This approach preserved result diversity while maintaining relevance.

Cost Management for AI-Based Enhancement. Integrating Claude AI for result enhancement introduced cost considerations. Batch enhancement was deliberately avoided, as it would increase unnecessary API usage. Instead, enhancement is user-driven: users manually select which retrieved text snippets to enhance. Global caching was implemented so that identical request bodies and queries reuse cached results, preventing redundant embedding generation and API calls. This design balances usability with cost efficiency.

6.2 Problem-Solving Approach

A systematic debugging methodology was adopted throughout development. Each pipeline component was tested independently prior to integration. Assumptions regarding data formats, embedding dimensions, and API behavior were explicitly verified. Fixes were applied incrementally, and validation layers were introduced to catch errors early. This approach minimized cascading failures and improved overall system robustness.

6.3 Lessons Learned

Several key insights emerged from the implementation:

- **Model consistency is critical**: Indexing and querying must use identical embedding models and configurations.
- **Context-aware preprocessing improves retrieval**: Intelligent HTML parsing significantly enhances chunk quality and semantic search performance.
- **Infrastructure decisions matter**: Scaling cloud resources can be more reliable than complex runtime optimizations in production environments.
- **Hybrid retrieval is more robust**: Combining semantic similarity with keyword relevance improves precision across diverse query types.
- **Manual evaluation remains essential**: Automated similarity scores alone do not fully capture retrieval quality.

6.4 Future Improvements

Future enhancements could further strengthen the system while maintaining scalability:

- **Automated relevance evaluation.** The current evaluation relies on manual relevance assessment, which limits both scale and statistical significance. An AI-based evaluator could automatically score thousands of retrieved results using the same ordinal relevance criteria, enabling continuous testing and more robust comparison between semantic and hybrid retrieval strategies as the system evolves.
- **Dynamic corpus expansion.** The system currently operates on a fixed corpus of pre-selected websites. Allowing users to submit new URLs would enable on-demand scraping, embedding, and indexing, transforming the system into a personalized research assistant. Computational costs could be managed through asynchronous processing and caching of previously ingested domains.

- **Hybrid weight optimization.** Hybrid retrieval currently uses fixed semantic and keyword weights selected heuristically. A systematic evaluation across multiple weight configurations could identify optimal balances for different query types. More advanced approaches could dynamically adjust weights based on query characteristics such as technical jargon, proper nouns, or quoted terms.

- **Query expansion with semantic awareness.** Retrieval recall could be improved by automatically expanding queries with semantically related terms or synonyms prior to embedding generation. This would reduce sensitivity to vocabulary variation and abbreviations, improving performance on queries that use informal or domain-specific language.

Acknowledgements

The author would like to thank Tsumbo Munyai for proposing the initial project concept and for providing guidance and feedback throughout the development of the project.