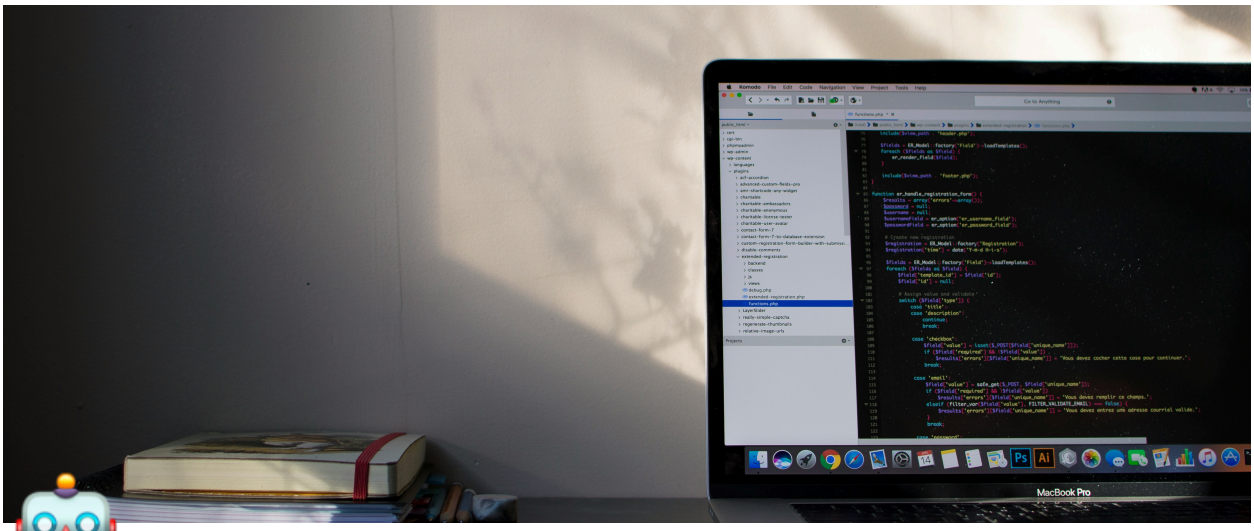

RAG System Documentation: The Batch QA Assistant

Objective

To design and implement a multimodal Retrieval-Augmented Generation (RAG) system that answers natural language questions using articles from **The Batch** newsletter by DeepLearning.AI, including text and associated images.

Approach Overview

The pipeline looks like:

1. **Data ingestion** → article_parser.py
2. **Preprocessing and chunking** → chunker.py
3. **Vector embedding and storage** → embedder.py
4. **RAG reasoning with Gemini** → rag_chain.py
5. **Interactive chat UI** → app.py (Streamlit)

Tools & Technologies

Component	Choice	Reasoning
LLM Embeddings	GoogleGenerativeAIEmbeddings	Native Gemini support for vector embeddings, the most popular and easy for embedding is the model embedding-001 from LangChain, but it's as gemini-embedding-exp-03-07
Vector Database	Chroma via langchain_chroma	Lightweight, efficient for local developing and using
LLM for QA	models/gemini-1.5-flash	It's easy, fast and for this task great option, Gemini is good for JSON outputs (there also could be ENUM mode which is well)
UI Framework	Streamlit	Easy in usage and fast in integrating
Parsing	BeautifulSoup, requests	Reliable web scraping with custom logic (takes HTML)
Chunking	RecursiveCharacterTextSplitter	Fine-tuned chunk creation with overlap for semantic continuity

Data Ingestion – article_parser.py

- **Data** are from The Batch archive from page.
- **Process**:
  - Pages 1–22 are scraped using a loop.
  - Embedded __NEXT_DATA__ JSON provides article metadata.
  - For each issue:
    - Individual article pages are fetched. (Each issue)
    - Titles, paragraph content, and associated images are extracted using BeautifulSoup. (h1, h2, h3, urls)
- And as an output I receive a JSON with issues (316) and titles in it. Example:

```
{
  "issue-301": {
    "title": "...",
    "url": "...",
    "articles": [
      {
        "title": "...",
        "content": "...",
        "image": "..."
      },
      ...
    ]
  },
  ...
}
```

Preprocessing – chunker.py

- Uses LangChain's RecursiveCharacterTextSplitter.
- Articles are split based on:
  - chunk_size=640
  - chunk_overlap=100
  - Threshold=500 to skip chunking for very short content.

  There are two options, if an article has a few words in content, then we just take it as one chunk, divide into chunks otherwise.
- Each chunk retains metadata:
  - Article title
  - Issue title
  - Issue and image URL

I have received **total number of chunks**: 4754.

Example if to take one of chunks:

```
{"id": "...",
"content": "...",
"metadata": {
   "article_title": "...",
   "issue_title": "...",
   "issue_url": "...",
   "image_url": "..."
}
```

Embedding & Indexing – embedder.py

- **Embeddings**:
  - Using Gemini embedding model: "models/embedding-001"
  - Task type: "SEMANTIC_SIMILARITY", because then we take the most similar to the query topics (vectors) and proceed working with them.
- **Storage**:
  - Embeddings and metadata are saved to a persistent Chroma DB at ./database/.
- **Integration**:
  - Documents are converted to LCDocument (LangChain format).
  - Re-usable vector_db(api_key) function builds and returns a search-ready vector database.

Question Answering Logic – rag_chain.py

answer_query_with_rag(query, api_key, vectordb)

1. **Retrieve Context**:
   - Finds top-k=3 relevant chunks from ChromaDB. (It’s okay to take more.)
2. **Prompt Gemini**:

- System prompt is embedded directly into a single "user" message (due to Gemini's API limitation).
- Gemini is instructed to respond using a strict JSON schema:

```
{
  "answers": [
    {
      "number": 1,
      "text": "...",
      "title": "...",
      "url": "...",
      "image_url": "..."
    }
  ]
}
```

- The context includes article metadata and content.

User Interface –app.py (with Streamlit)

- Supports:
  - API key entry for a customer
  - Chat-style question input
  - Persistent vector DB in st.session_state.vector_database (Is initialized only once when user enters the page, then RAG just work with it - to ensure always taking fresh information)
  - Answer rendering in markdown with image previews

Evaluation Strategy

The system was evaluated using:

- **Manual QA Checks**: answers align with retrieved context.
- **Structured Output Validity**: JSON schema is correctly followed.

- **Faithfulness**: Answers come solely from provided context.
- **Relevance**: Retrieved articles match query intent.
- **RAGAS results**:
    - **Context precision (1.00)** and **context recall (1.00)**, indicating that the retrieved documents were highly relevant and fully covered the necessary ground truth information. The **answer relevancy score (0.88)** also confirms that the generated responses closely aligned with the user's questions.
    - **faithfulness score (0.56)** and **answer correctness score (0.49)** suggest that the model occasionally generated information not strictly supported by the context, or omitted key factual elements. These scores highlight opportunities to improve prompt engineering, context formatting, or model calibration for more grounded, accurate outputs. (for improvements then 😉)

Usage Guide

demo-video

> https://drive.google.com/file/d/14DgQkKIFMBZIeDk6hfSNwmUYWE3VFeOF/view?usp=sharing

Requirements

Install dependencies via:

```
pip install -r requirements.txt
```

Run app

```
streamlit run app.py
```

Set API key

You’ll be prompted to enter your Gemini API key on first use.

And here you are, you freely can ask questions.