Book Worm: A RAG Implementation

Julian Rosas

DSC 599: Data Science Capstone

December 11, 2025

Contents

Abstract

This report presents Book Worm, a Retrieval-Augmented Generation (RAG) system designed specifically for multi-series books with guard rails for spoiler prevention. The app utilizes the Fandom MediaWiki API to extract and process content from multiple book series, utilizes text chunking algorithms, stores data in a ChromaDB vector database for semantic search, and uses Ollama 3.2 for answer generation. The app is used by the user providing a book series filter (optional), providing a question, the vector database is queried based on semantic retrieval, then the chunks are input to Ollama to generate and provide answers. The current implementation offers a Streamlit deployable user interface and offers easy additions for new book series for further information expansion. Future additions will include spoiler tags for chunks that include major spoilers. The book series successfully processed so far include the Red Rising and Harry Potter series with 853 total text chunks, achieving effective semantic retrieval while maintaining spoiler-free user experiences based on reading progress.

1 Introduction

Long-running book series create issues with having to remember so many characters or events that are re-introduced as the series progresses. Often, one might forget a certain character and what kind of impact they have had on the story thus far. Current solutions call for just looking up the character to read a quick description on their background and influence on the story so far, but this often include spoilers for major plot point including the worst kind of spoilers, deaths. These spoilers can simply come in the form of suggested searches right underneath a completely separate specific question you may have asked.

This work introduces Book-Worm, a specialized RAG system that addresses these challenges through several key contributions:

1. **Multi-Series Architecture**: A modifiable configuration system supporting multiple book series with different structures
2. **Smart Chunking**: Efficient text segmentation algorithms optimized for narrative content
3. **Automated Data Pipeline**: End-to-end processing from wiki sources to query-able vector embeddings
4. **Context-Aware LLM Integration**: LLM response generation that synthesizes retrieved information while respecting spoiler constraints and reading progress
5. **Streamlit Deployment**: Deployment through Streamlit to provide an easy-to-use interface

The current implementation demonstrates practical applicability using long-running series including Red Rising by Pierce Brown and Harry Potter by J.K. Rowling, with support for easy additions for other series.

2 Related Work

2.1 Book QA: Stories of Challenges and Opportunities

Angelidis et al. (2019) introduced BookQA, a question-answering research system that implemented AI models to answer questions given a full book. It uses a retrieval and reasoning pipeline. The retriever finds passages in a book that are most relevant to a question (using Bert-based embedding

models). Then a memory network is used to read the passages and perform multi-hop reasoning to infer an answer. The implementation was tested on the NarrativeQA dataset. Book Worm differs from this work through a retrieval-augmented generation implementation where you can set up a library of multiple books from different series through the Fandom API. The RAG implementation allows for spoiler-aware answers generated by an LLM vs the BookQA implementation that focuses on retrievals for one specific query. Book Worm offers more open-ended queries with constraints to limit spoiling.

2.2 Frustratingly Hard Evidence Retrieval for QA Over Books

Mou et al. (2020) further worked n book QA by presenting a system that tackles the challenge of QA over full-length books. It worked by receiving a question over a long book, then the system retrieves supporting evidence over the book and generates an answer. They experimented with BERT or GPT-2 for retrieval and reading but struggled with evidence retrieval, especially given long sources like an entire book. Even when increasing the number of passages retrieved, the model did not always perform better. This is very similar to my approach, but I am again using the Fandom API so questions might be more limited. But Book Worm is meant for general overviews to assist readers as they are progressing through a story rather than specific answers. This also allows for less data processing, but also makes it more difficult to ensure an answer is being answered correctly if it does become specific.

3 Proposed Approach

3.1 Data Extraction and Processing

The system utilizes the Fandom MediaWiki API to extract structured content from series-specific wikis. The extraction process involves:

1. **API Configuration**: Series-specific wiki endpoints and page titles
2. **WikiText Parsing**: Conversion of MediaWiki markup to clean text
3. **Section Extraction**: Hierarchical content organization by page sections

4. **Metadata**: Addition of series, book number, and character information

3.2 System Architecture Overview

The Book-Worm system follows a comprehensive RAG architecture as illustrated in Figure 1:

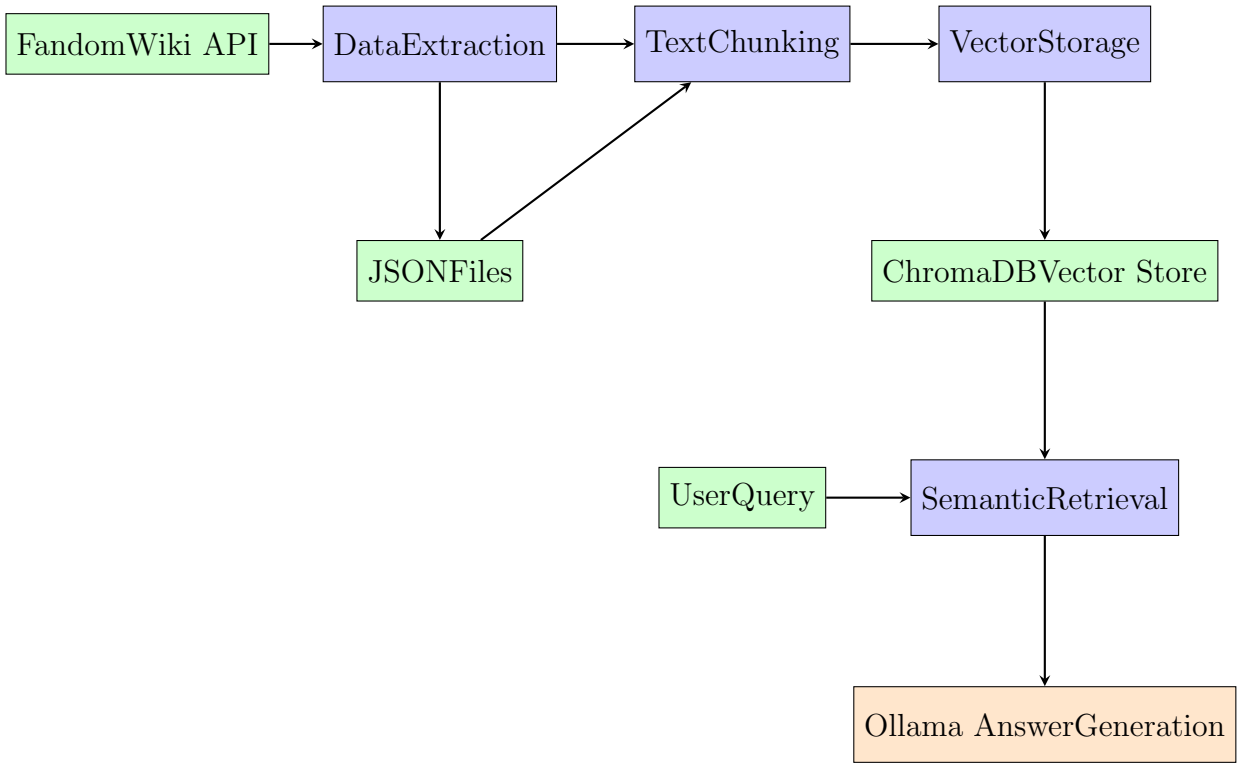

Figure 1: Book-Worm RAG System Architecture

3.3 Text Chunking Algorithm

The chunking algorithm implements a section-based approach to optimize content for vector embeddings while preserving narrative coherence:

```
def merge_chunks_by_section(chunks, name, doc_type,
    series_name, max_words=400):
    # Group text by section headers
    section_map = defaultdict(list)
```

```
    for chunk in chunks:
        section_map[chunk['section']].append(chunk['text'])

    merged = []
    for section, paragraphs in section_map.items():
        # Combine all paragraphs in a section
        block = '\n\n'.join(paragraphs)
        words = block.split()

        # Split into fixed-size chunks of max_words
        for i in range(0, len(words), max_words):
            chunk_text = ' '.join(words[i:i+max_words])
            chunk_data = {
                'text': chunk_text,
                'section': section,
                'name': name,
                'type': doc_type,
                'series': series_name
            }
            merged.append(chunk_data)

    return merged
```

Listing 1: Chunking Algorithm Implementation

This algorithm:

- Groups paragraphs by section headers to preserve context
- Splits text into fixed chunks of 400 words maximum
- Preserves metadata (section, name, type, series) for each chunk
- Ensures consistent chunk sizes for optimal vector embeddings

4 Implementation

4.1 Technology Stack

The implementation utilizes the following technologies:

- **Python 3.11**: Core implementation language
- **ChromaDB**: Vector database for semantic search

- **Requests**: HTTP client for API interactions
- **Regular Expressions**: Text cleaning and parsing
- **JSON**: Intermediate data storage format
- **Ollama 3.2**: Answer generation through Ollama 3.2

4.2 Series Configuration System

The system employs a declarative configuration approach for adding new series:

```
SERIES_CONFIG = {
    "Red Rising": {
        "wiki": "red-rising.fandom.com",
        "pages": [
            {"title": "Darrow_O'Lykos",
             "type": "characters",
             "name": "Darrow O'Lykos"},
            {"title": "Virginia_au_Augustus",
             "type": "characters",
             "name": "Mustang"},
            {"title": "The_Conquering",
             "type": "events",
             "name": "The Conquering"},
            {"title": "The_Institute",
             "type": "locations",
             "name": "The Institute"}
        ]
    },
    "Harry Potter": {
        "wiki": "harrypotter.fandom.com",
        "pages": [
            {"title": "Harry_Potter",
             "type": "characters",
             "name": "Harry Potter"},
            {"title": "Hermione_Granger",
             "type": "characters",
             "name": "Hermione Granger"}
        ]
    }
}
```

Listing 2: Series Configuration Example

5 Results

The current implementation successfully processes multiple book series with the following metrics:

Series	Characters	Events	Locations	Total Chunks
Red Rising	37	1	1	471
Harry Potter	3	0	0	382
Total	**40**	**1**	**1**	**853**

Table 1: Current System Processing Results

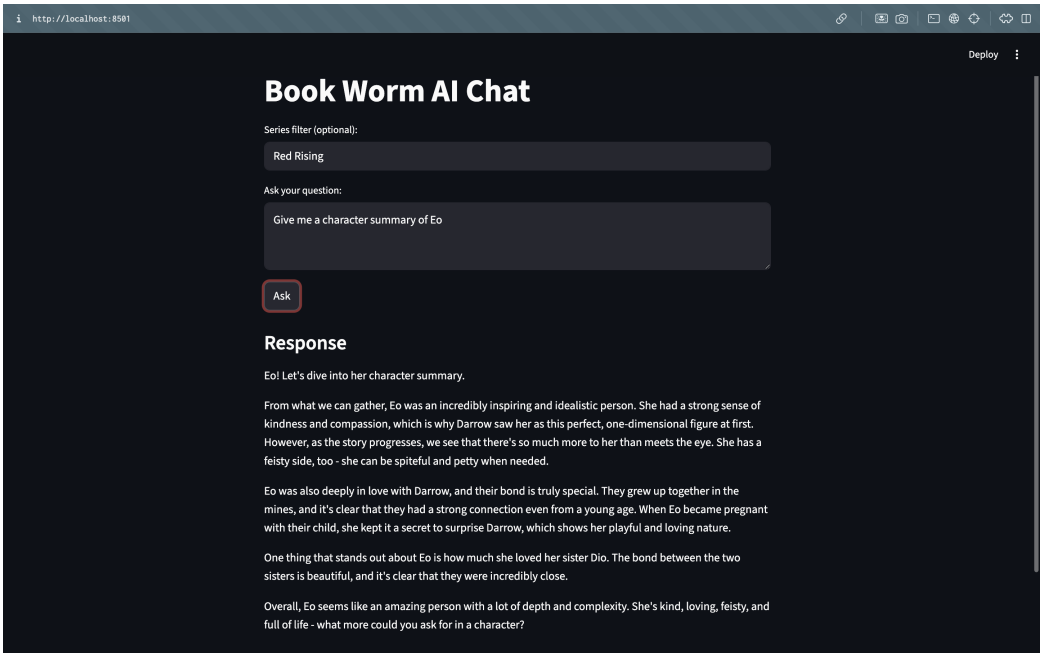

Figure 2: Character Query Results

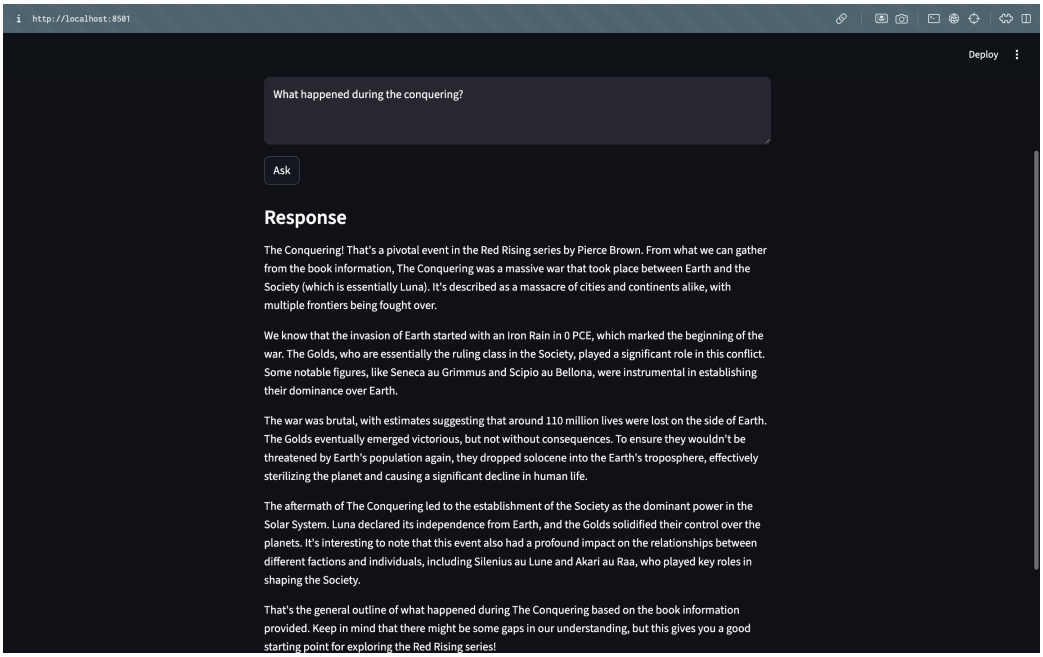

Figure 3: Event Query Results

6 Future Work

1. **Spoiler Tag**: Addition of spoiler tags for retrieved chunks with spoilers
2. **Additional Series**: New additions for any series requested for any user or myself

7 Conclusion

This report showed the almost completed version of Book-Worm, a RAG system designed for multi-series books with guard rails for spoilers. Balancing spoiler limitations with proven to be difficult with the implementation of an LLM with guidelines, but have been getting consistent results for test queries for accurate retrievals and spoiler avoidance. Still need to perform further testing and add a spoiler tag for retried chunks to ensure correct answers are being retrieved while the strict spoiler filtering process occurs. Future work required is an the spoiler tag, and a potential for awareness for plot twists recognition, which may prove to be difficult.

References

[1] Angelidis, S., Suhara, Y., Kannan, R., McAuley, J., & Bansal, M. (2019). *BookQA: Stories of Challenges and Opportunities*. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (pp. 693–703). Association for Computational Linguistics. Retrieved from https://aclanthology.org/P19-1067

[2] Mou, L., Wang, R., Li, G., Meng, L., & Song, Y. (2020). *Frustratingly Hard Evidence Retrieval for QA Over Books*. In Proceedings of the 2nd Workshop on Narrative Understanding (pp. 79–84). Association for Computational Linguistics. Retrieved from https://aclanthology.org/2020.nuse-1.13