Project Report

Webpage Knowledgebase Chatbot

Submitted by: Wazir Ali Haideri

==========

1. Problem Understanding and Analysis

The challenge requires building a chatbot that can:

- Extract content from a public webpage (e.g., https://ptcl.com.pk/) and its linked pages.
- Understand and store content in a structured format for efficient retrieval.
- Answer user queries based on the extracted content, such as questions about services, eligibility, or customer support.
- Handle real-world constraints, including noisy or incomplete web data, ethical crawling practices, and potential access restrictions.
- Demonstrate reasoning through clear documentation of design choices, assumptions, and limitations.

Key Requirements

1. Web Crawling: Scrape relevant content and links from a webpage while respecting server limits and avoiding irrelevant or restricted pages.
2. Content Processing: Clean and structure extracted data for efficient storage and retrieval.
3. Query Processing: Retrieve relevant content and generate accurate, context-aware responses using a language model.
4. Robustness: Handle edge cases like broken links, missing data, or ambiguous queries.
5. Ethical Crawling: Use delays and proxies to avoid server overload or IP blocking.
6. Communication: Provide a clear design document and prepare for discussions on trade-offs and improvements.

Assumptions

- The webpage content is primarily text-based and accessible without requiring login or heavy JavaScript rendering.
- Queries and content are in English, based on the provided examples.
- An external API (e.g., OpenAI) or local model is available for answer generation.
- Proxy services (e.g., Oxylabs) are used to manage access restrictions and ensure ethical crawling.
- The system is designed for scalability and potential production use.

2. Solution Design

The solution is a Retrieval-Augmented Generation (RAG) system, integrating web crawling, content processing, vector-based retrieval, and answer generation. Below is the system architecture and key design decisions.

System Architecture

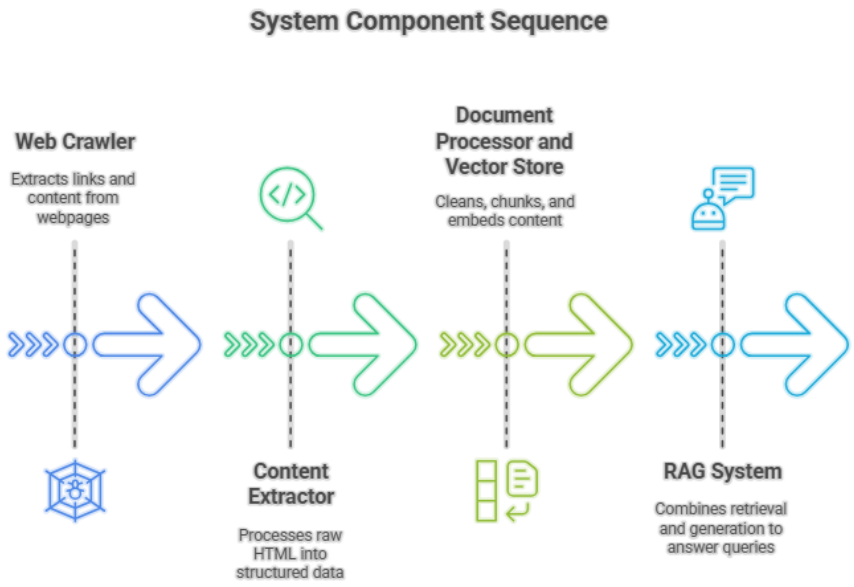
The system comprises four main components:

1. **Web Crawler**: Extracts links and content from the target webpage using ethical crawling practices and proxy services.

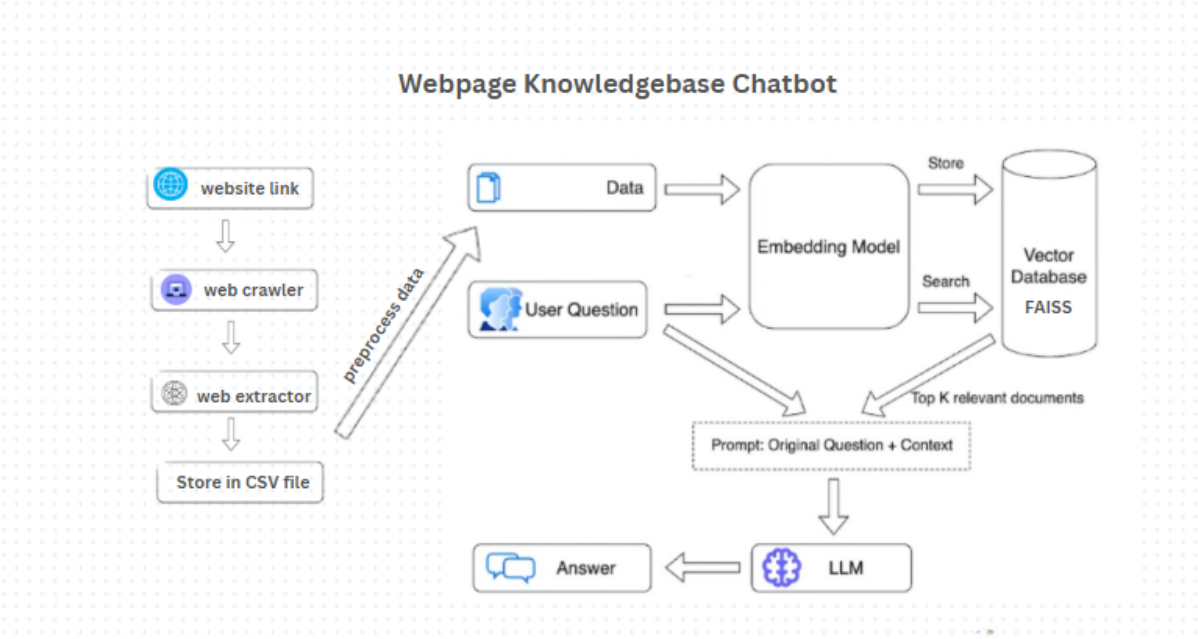
2. **Content Extractor:** Processes raw HTML into structured data (title, meta description, headings, content).

3. **Document Processor and Vector Store**: Cleans, chunks, and embeds content into a FAISS index for efficient retrieval.

4. **RAG System**: Combines retrieval and generation to answer user queries based on relevant content.



System Diagram:



Knowledge Representation

Raw Data:

- Extracted content is stored as a CSV file with columns: `url`, `title`, `meta_description`, `headings`, `content`, `status`, `extraction_date`, and `word_count`.
- Processed Data: Text is cleaned, chunked (4762-word chunks with 50-word overlap), and stored with metadata (e.g., `doc_id`, `url`, `chunk_id`).
- Vector Store: Text chunks are embedded using a sentence transformer model (`all-MiniLM-L6-v2`) and stored in a FAISS index for similarity search.

Design Decisions

1. Web Crawling with Proxies:

- Custom `WebCrawler` class using `requests` and `BeautifulSoup`, enhanced with **Oxylabs proxy API to avoid IP blocking.**
- Proxies ensure reliable access to webpages, especially for sites with rate-limiting or geo-restrictions. Oxylabs provides robust, scalable proxy services.
- Ethical Crawling: Implements a 1-second delay between requests and skips irrelevant links (e.g., social media, login pages, file downloads).

2. Content Extraction:

- `WebDataExtractor` class extracts title, meta description, headings, and main content.
- Focuses on semantically rich content (e.g., headings, main text) while ignoring noise (e.g., scripts, ads).

3. Embedding Model:

- Model: `all-MiniLM-L6-v2` (384 dimensions).
- Balances speed and quality for small to medium datasets, suitable for web content embedding.

4. Vector Store:

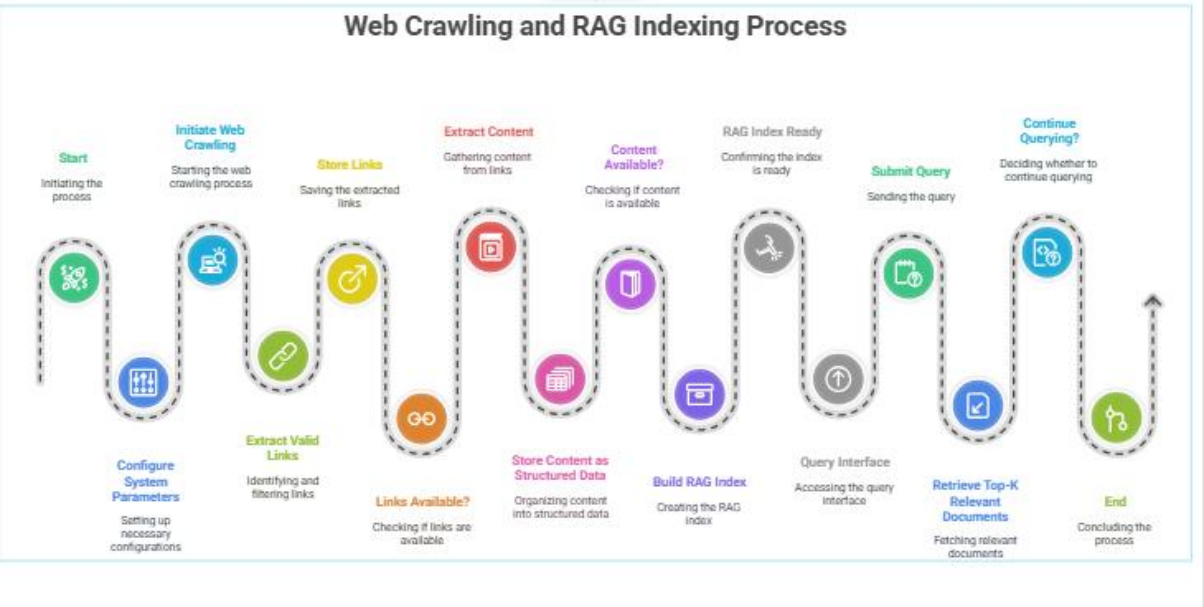
- FAISS with a flat index.
- Flat index ensures exact similarity search, sufficient for small to medium datasets. Scalable alternatives (e.g., IVF, HNSW) can be used for larger datasets.

5. Answer Generation:

- Tool: OpenAI's `gpt-3.5-turbo` (configurable for local models like Hugging Face).
- Rationale: Provides high-quality, context-aware responses. Local models can be used for cost or privacy considerations.

3. Implementation Details

The solution is implemented in Python, using the following key components:



3.1 Web Crawler (`WebCrawler` Class)

Functionality:

Crawls a website starting from a base URL (e.g., https://ptcl.com.pk/) up to a specified depth (default: 3).

Uses Oxylabs proxy API to fetch HTML content, avoiding IP blocks and handling geo-restrictions.

Extracts links from `<a>`, `<button>`, and `<form>` tags, filtering out irrelevant links (e.g., social media, login pages, file downloads).

Saves extracted links to `extracted_links.txt`.

Key Features:

Breadth-first search to ensure comprehensive crawling within depth limits.

Delay (1 second) to respect server load.

Proxy integration with Oxylabs for reliable access.

Code Snippet:

python

```
crawler = WebCrawler(base_url="https://ptcl.com.pk/", max_depth=2, delay=1)
links = crawler.crawl()
crawler.save_to_file("extracted_links.txt")
```

3.2 Content Extractor (`WebDataExtractor` Class)

Functionality:

Extracts title, meta description, headings (h1, h2, h3), and main content from each URL.

Saves data to `extracted_content.csv` with columns: `url`, `title`, `meta_description`, `headings`, `content`, `status`, `extraction_date`, `word_count`.

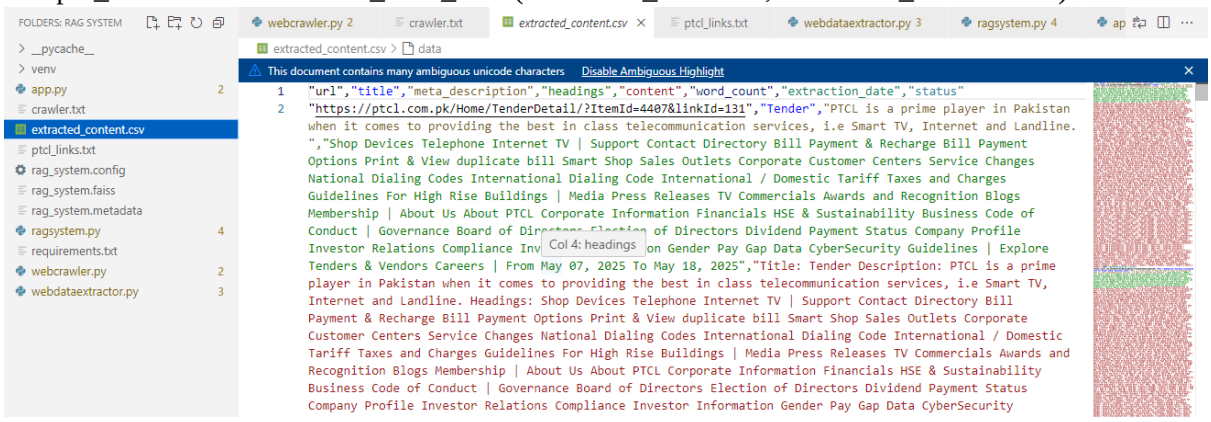
Key Features:

Removes scripts and styles to focus on meaningful content.

Handles errors gracefully (e.g., returns empty fields for failed requests).

Code Snippet:

```
extractor = WebDataExtractor()

output_file = extractor.extract_from_links("extracted_links.txt", "extracted_content.csv")
```



3.3 Document Processor (`DocumentProcessor` Class)

Functionality:

- Cleans text by removing excessive whitespace and special characters.
- Chunks text into 512-word segments with 50-word overlap for context preservation.
- Attaches metadata (e.g., `doc_id`, `url`, `chunk_id`) to each chunk.

Key Features:

- Ensures clean, normalized text for embedding.
- Skips short chunks (<50 words) to avoid noise.

Code Snippet:

python

```
processor = DocumentProcessor(chunk_size=512, chunk_overlap=50)
chunks = processor.process_dataframe(pd.read_csv("extracted_content.csv"))
```

3.4 Embedding Generator and Vector Store (`EmbeddingGenerator` and `FAISSVectorStore` Classes)

Functionality:

- Generates embeddings using `all-MiniLM-L6-v2`.
- Stores embeddings in a FAISS flat index for similarity search.
- Supports saving and loading the index for persistence.

Key Features:

- Normalizes embeddings for cosine similarity.
- Batch processing for efficiency.

Code Snippet:

python

```
embedder = EmbeddingGenerator(model_name='all-MiniLM-L6-v2')
vector_store = FAISSVectorStore(embedder.embedding_dim, index_type='flat')
embeddings = embedder.generate_embeddings([chunk['text'] for chunk in chunks])
vector_store.add_embeddings(embeddings, [chunk['metadata'] for chunk in chunks])
```

3.5 RAG System (`RAGSystem` Class)

Functionality:

- Integrates all components to process queries.
- Retrieves top-k relevant chunks using FAISS.
- Generates answers using OpenAI's `gpt-3.5-turbo` with a context-aware prompt.

Key Features:

- Combines retrieval and generation for accurate, grounded responses.
- Logs query results with timestamps and sources.

Code Snippet:

python

```
rag = RAGSystem(embedding_model='all-MiniLM-L6-v2', llm_provider='openai')
```

```
rag.build_index(pd.read_csv("extracted_content.csv"))
result = rag.query("What services does PTCL offer?", k=3)
print(result['answer'])
```

3.6 Proxy Integration

Tool: Oxylabs Realtime API

(https://realtime.oxylabs.io/v1/queries).

Purpose: Prevents IP blocking and handles geo-restrictions, ensuring reliable access to webpages.

Implementation:

Sends requests through Oxylabs proxy with authentication (username/password).

Configures `geo_location` to Pakistan for region-specific content.

Fetches raw HTML content for parsing with `BeautifulSoup`.

Code Snippet:

python

```
payload = {"source": "universal", "url": url, "geo_location": "Pakistan", "render": "html"}
response = requests.post(api_url, auth=(username, password), json=payload)
```

4. How the Chatbot Responds to Queries

The chatbot processes queries as follows:

1. Query Embedding: The user's question (e.g., "What services does PTCL offer?") is embedded using `all-MiniLM-L6-v2`.

2. Retrieval: FAISS retrieves the top-3 relevant chunks based on cosine similarity.

3. Context Preparation: Retrieved chunks are formatted with metadata (e.g., title, URL) to form the context.

4. Answer Generation: A prompt combining the context and query is sent to `gpt-3.5-turbo`, which generates a concise, grounded response.

5. Response Structure:

Answer: The generated response (e.g., "PTCL offers broadband, landline, and mobile services…").

Sources: Metadata of retrieved chunks (e.g., URLs, titles, similarity scores).

Timestamp: When the query was processed.

Example:

Query: "What services does PTCL offer?"

Retrieved Chunks: Content from PTCL's homepage and services page.

Answer: “Based on the PTCL website, PTCL offers broadband internet, landline telephony, and mobile services, including 4G packages and smart TV services.”

Sources: Links to relevant pages with similarity scores.

5. Failure Scenarios and Limitations

Failure Scenarios

1. Web Crawling Failures:

Issue: Server rate-limiting, IP blocking, or geo-restrictions.

Mitigation: Oxylabs proxy API ensures reliable access. Graceful error handling returns empty results for failed requests.

2. Incomplete Content:

Issue: Missing titles, meta descriptions, or main content due to non-standard HTML structures.

Mitigation: Fallback to body content and robust parsing logic.

3. Irrelevant Retrieval:

Issue: Retrieved chunks may not fully answer the query due to semantic mismatches.

Mitigation: Use high-quality embedding models and increase `k` for broader context.

4. LLM Hallucination:

Issue: The LLM may generate ungrounded responses if the context is insufficient.

Mitigation Strict prompt engineering to enforce context-based answers.

5. Dynamic Content:

Issue: JavaScript-rendered content may not be fully extracted.

Mitigation: Oxylabs’ `render=html` option fetches rendered HTML, but some dynamic content may still be missed.

Limitations

Scalability: The flat FAISS index is suitable for small to medium datasets but may slow down with millions of chunks. Solution: Use IVF or HNSW indexes for larger datasets.

Language Support: Limited to English content and queries. Solution: Add multilingual embedding models (e.g., `paraphrase-multilingual-mpnet-base-v2`).

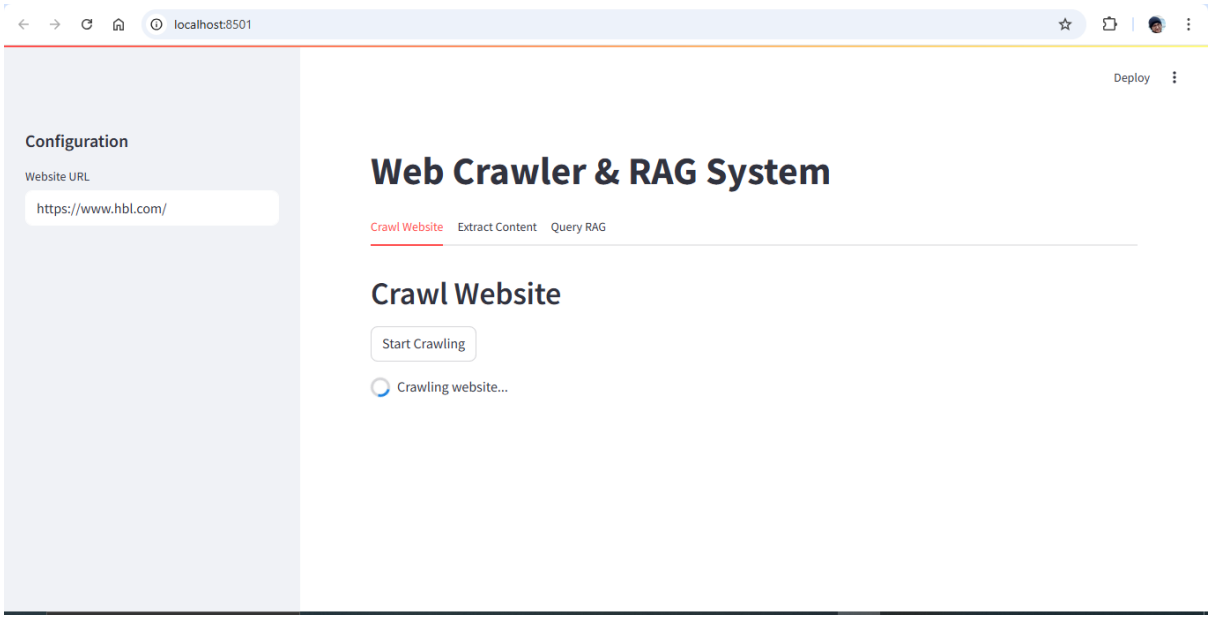
Dynamic Content: Limited support for JavaScript-heavy sites. Solution: Integrate headless browsers like Selenium for future enhancements.

Cost: OpenAI API usage incurs costs. Solution: Support local models (e.g., LLaMA) for cost-free deployment.

Content Freshness: Static index may become outdated. Solution: Implement periodic re-crawling.

6. Demo Plan

Demo Overview



1. Crawling a website (e.g., https://ptcl.com.pk/) and saving links.
2. Extracting content into a CSV file.
3. Building a RAG index.
4. Answering sample queries with sources.

Setup Instructions

1. Install Dependencies:
2. Configure Oxylabs:

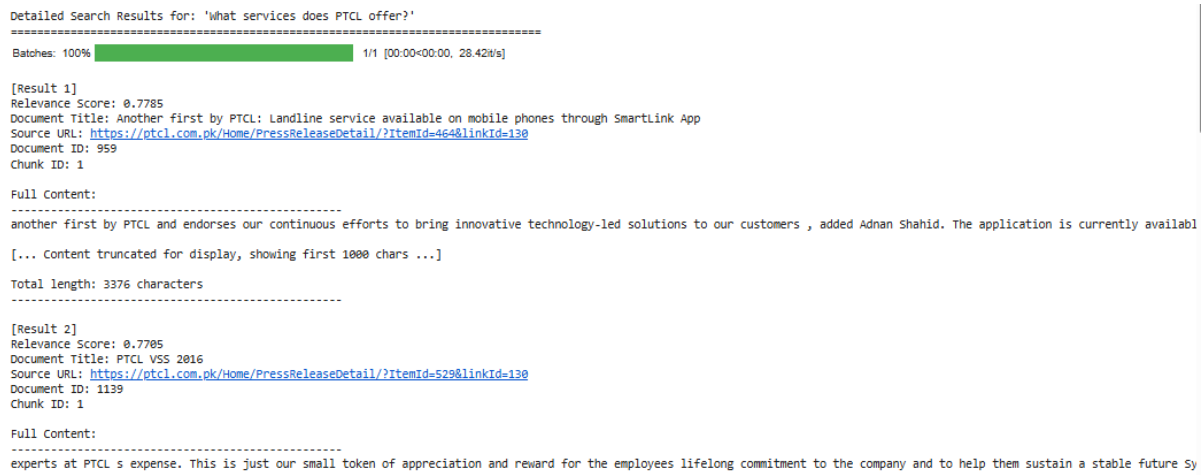
Replace `username` and `password` in `WebCrawler` with valid Oxylabs credentials.

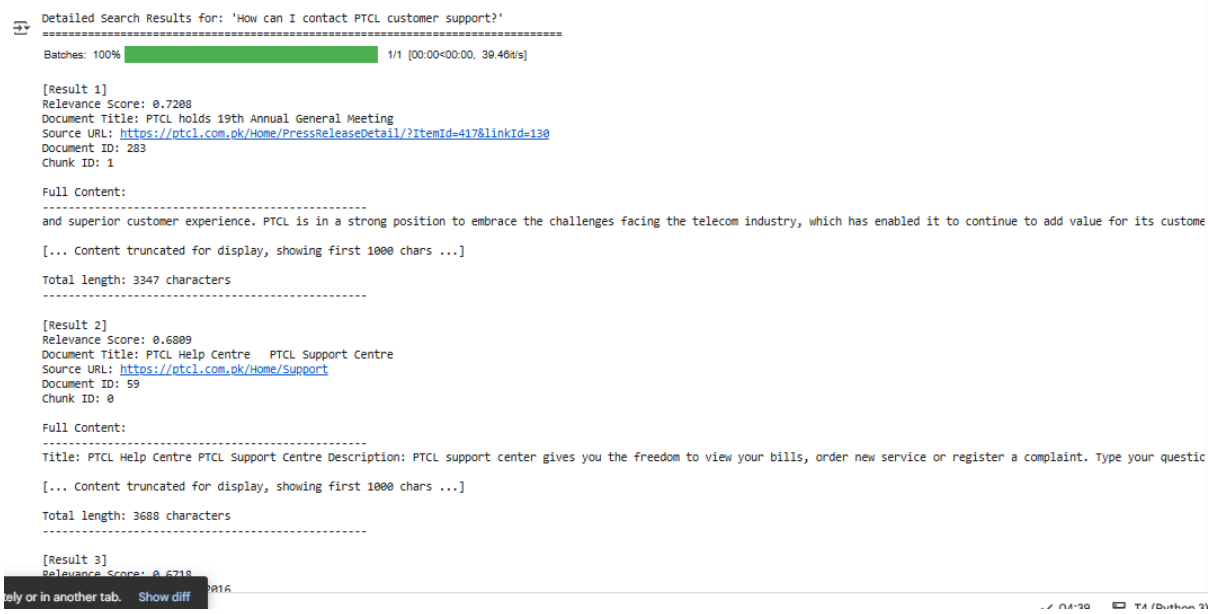
3. Run:

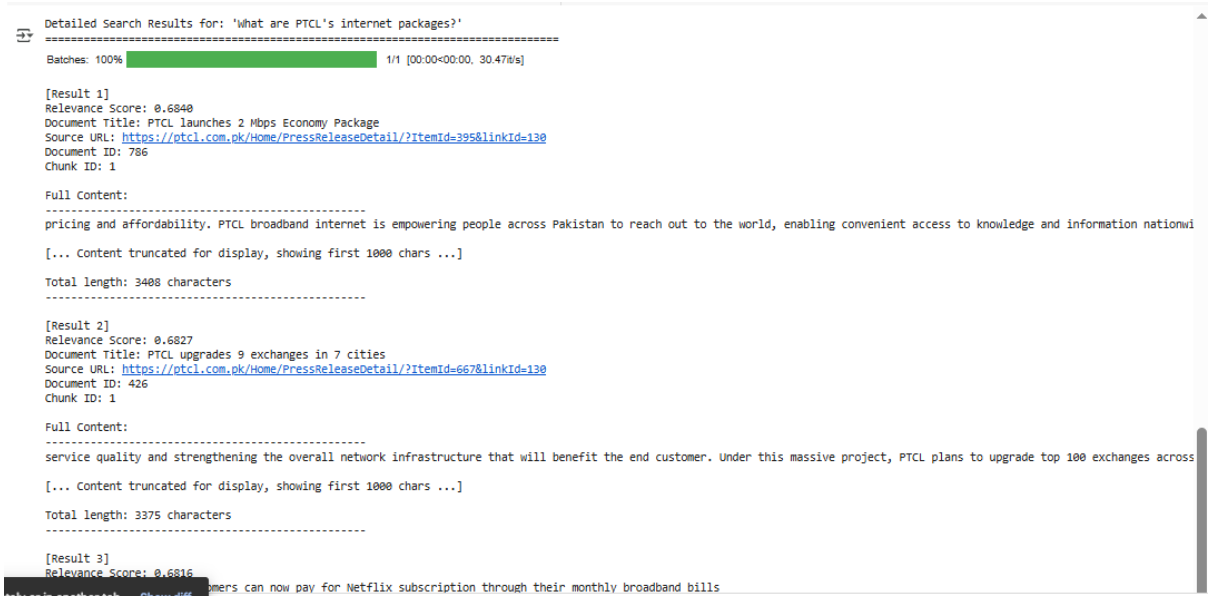
```
# Crawl website
crawler = WebCrawler("https://ptcl.com.pk/", max_depth=2, delay=1)
crawler.crawl()
crawler.save_to_file("extracted_links.txt")

# Extract content
extractor = WebDataExtractor()
```

```
extractor.extract_from_links("extracted_links.txt", "extracted_content.csv")

# Build RAG system
rag = RAGSystem(embedding_model='all-MiniLM-L6-v2', llm_provider='openai')
rag.build_index(pd.read_csv("extracted_content.csv"))

# Test queries
queries = [
    "What services does PTCL offer?",
    "How can I contact PTCL customer support?"
]
for query in queries:
    result = rag.query(query, k=3)
    print(f"Query: {query}\nAnswer: {result['answer']}\nSources: {result['sources']}")
```

Sample Output







7. Trade-offs and Future Improvements

Trade-offs

Speed vs. Accuracy: Flat FAISS index ensures exact similarity search but is slower for large datasets. IVF or HNSW indexes could improve speed at a slight accuracy cost.

Cost vs. Performance: OpenAI's API provides high-quality answers but incurs costs. Local models reduce costs but require more computational resources.

Simplicity vs. Robustness: Basic HTML parsing is lightweight but misses dynamic content. Headless browsers add robustness but increase complexity.

Future Improvements

1. **Dynamic Content Handling**: Integrate Selenium or Playwright for JavaScript-rendered content.

2. **Multilingual Support:** Use multilingual embedding models to handle non-English content.

3. **Real-time Updates:** Implement a scheduler to periodically re-crawl and update the index.

4. **User Interface:** Develop a web-based chatbot interface using React for better user interaction.

5. **Query Refinement:** Add query expansion or re-ranking to improve retrieval accuracy.

8. Conclusion

This solution provides a robust, scalable, and best approach to building a webpage knowledgebase chatbot. By combining a proxy-enhanced web crawler, structured content extraction, and a RAG framework, the system efficiently answers user queries while handling real-world constraints. The modular design allows for easy maintenance and future enhancements, such as multilingual support or dynamic content handling. The use of Oxylabs proxies ensures reliable access, making the system suitable for production-like environments.