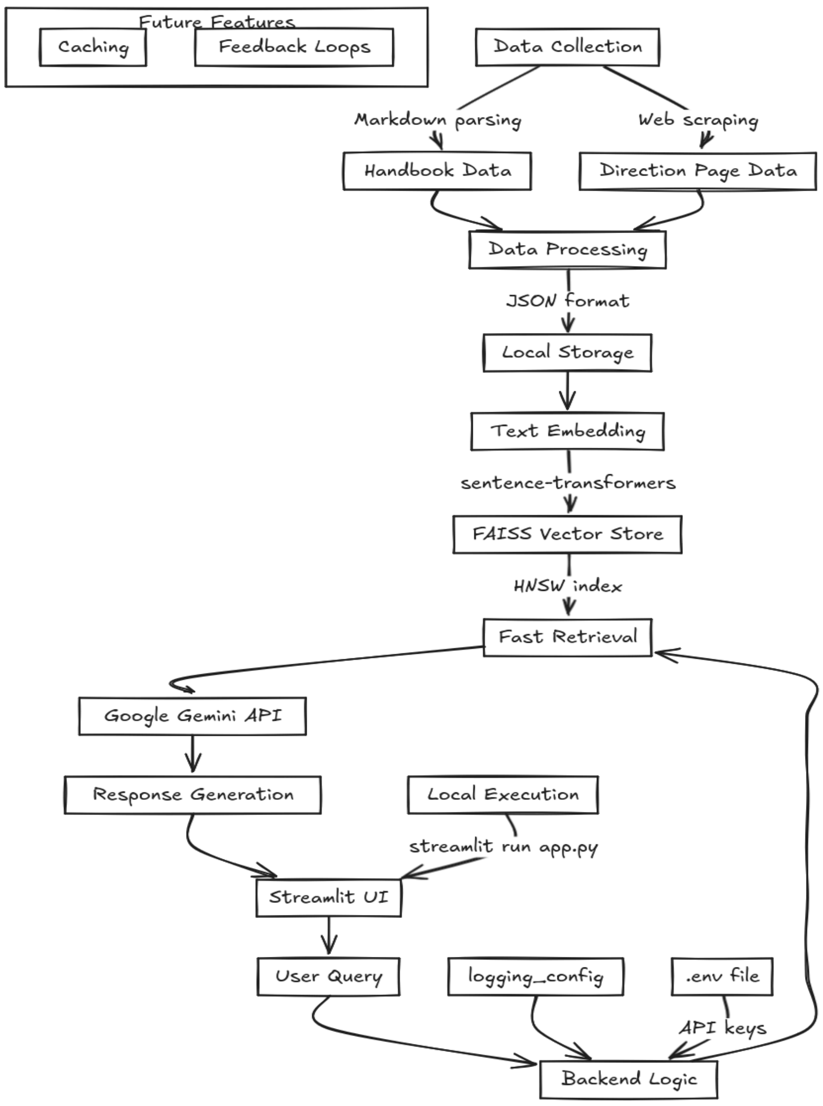
Project Documentation

Project Overview

This project is a chatbot designed to utilize context from GitLab's Handbook and Direction pages to answer user queries. The main objective was to create an interactive, AI-powered chatbot that retrieves relevant information using embeddings and generates responses via the Google Gemini API. The project incorporates advanced NLP techniques to preprocess, embed, and efficiently retrieve context.

Flow:



Approach and Key Decisions

1. Data Collection and Parsing

Gathering data from GitLab's Handbook and Direction pages was the initial focus:

- **GitLab Handbook**: With over 3,000 pages and approximately 3.9 million words, parsing the HTML content directly would be a complex and time-consuming task. Therefore, I decided to extract data from the Markdown files, which have a simpler structure and are easier to parse.
- **GitLab Direction Page**: This data was parsed from an HTML page using web scraping techniques to capture structured content.

Key Technical Choices:

- **BeautifulSoup**: Used for its simplicity and flexibility in parsing HTML content.
- **MarkdownIt**: Selected for parsing Markdown files due to its efficient tokenization and capability to handle nested structures and headers.

2. Data Transformation to JSON

Parsed data from both sources was transformed into a uniform JSON format for easy handling and consistency:

- **Structure**: Each JSON entry includes `header` and `content` fields.

Scripts:

- `data_retrieval_handbook.py`: Processes Markdown files to extract and convert content into structured JSON.
- `data_retrieval_direction.py`: Scrapes and parses the Direction page, generating JSON output.

Decisions:

- **Normalization**: Headers starting with HTML-like tags (`<`) were filtered out, and only non-empty fields were retained to ensure data quality.
- **Structure Simplicity**: While experimenting with a hierarchical structure where documents referenced parent headers, this approach added complexity and reduced the accuracy of context retrieval. Thus, a simpler structure was chosen.

3. Embeddings and Vector Store Creation

Semantic search was facilitated by embedding the parsed data into a vector space:

- **Model Used**: `sentence-transformers/all-MiniLM-L6-v2`, chosen for its optimal balance between performance and speed.
- **Vector Storage**: FAISS (Facebook AI Similarity Search) was used for fast vector lookup and retrieval.

Key Technical Steps:

- **Embedding Precomputation**: Embeddings were generated in parallel using `ThreadPoolExecutor` to improve processing speed.
- **Vector Store Optimization**: Configured with an HNSW (Hierarchical Navigable Small World) index to enhance search speed and accuracy.
- **Local Storage**: The indexed vector store was saved locally, reducing the need to re-index data upon restarting the app.

4. Integration with Streamlit

Streamlit was chosen for developing the web interface due to its ease of use and rapid prototyping capabilities. The app:

- Accepts user queries.
- Retrieves relevant context from the vector store.
- Displays responses based on the retrieved data.

Deployment:

- The app can be run locally with `streamlit run app.py`, making it available at `http://localhost:8501`.

5. API Integration with Google Gemini

To generate high-quality responses, the chatbot was integrated with the **Google Gemini API**:

- **Configuration**: Environment variables, managed using `dotenv`, load the API key securely.
- **Interaction**: Formatted prompts with relevant context are sent to Gemini, ensuring that responses are based on retrieved data only.

6. Logging and Monitoring

A custom `logging_config` module was implemented to monitor:

- Data parsing steps.
- Embedding and vector store creation.
- Query processing and response generation.

This logging setup aids in debugging and maintaining consistent performance.

Key Technical Choices

Choice of Libraries and Tools

- **BeautifulSoup**: Reliable for web scraping and parsing HTML.
- **MarkdownIt**: Lightweight and effective for parsing Markdown files.
- **FAISS**: Essential for efficient similarity search and vector retrieval.
- **HuggingFace Embeddings**: High-quality embeddings suitable for semantic search.
- **Streamlit**: Ideal for building quick, interactive web interfaces.
- **Google Gemini API**: Provides natural, human-like chatbot responses.

Design Considerations

- **Modular Design**: Ensured modularity for easy updates and scalability.
- **Data Structure**: The decision to use JSON format allows for flexibility and integration with other tools.
- **Parallel Processing**: Used for faster embedding of large datasets.
- **Security**: Sensitive data, like API keys, is stored securely using `.env` files.

Challenges and Solutions

Parsing Complex Markdown

The GitLab handbook has complex, nested structures. Capturing meaningful content while managing various heading levels was challenging.

Solution: `MarkdownIt` provided custom parsing logic and allowed for filtering out irrelevant data elements effectively.

Large-Scale Embedding and Search

Handling and searching through large-scale text data posed significant performance challenges.

Solution: The `all-MiniLM-L6-v2` model was chosen for its balance between size and accuracy. FAISS's HNSW index was configured to optimize vector search, balancing search speed and recall.

Future Enhancements

- **Caching**: Implement caching for frequently asked queries to save resources and response time.
- **Feedback Loop**: Integrate a feedback mechanism for users to rate responses, improving accuracy over time.
- **UI Enhancements**: Improve the app's user interface for a more intuitive experience.
- **Cloud Deployment**: Plan to deploy the application on cloud platforms for broader access and scalability.

Learning

Technical Learnings

- **Data Structuring**: I have never worked with this huge amount of data hence handling large, unstructured data and converting it into structured formats was invaluable for improving data processing skills and it helped me gain a deeper understanding of data manipulation.
- **Optimizing Semantic Search**: Working with vector embeddings and configuring FAISS indexing taught the importance of balancing accuracy and speed in search tasks.
- **Modular Codebase**: Designing a modular codebase proved beneficial for maintaining, testing, and scaling the application.

Personal Learnings

- **Decision Making**: Learned the importance of making informed choices based on project requirements (e.g., choosing to parse Markdown over HTML).
- **Problem Solving**: Developing solutions to parsing nested data structures and embedding large datasets deepened problem-solving skills.
- **Performance Tuning**: Optimizing the embedding and retrieval pipeline highlighted the importance of efficient resource management in large-scale applications.

Conclusion

This project demonstrates a robust approach to building an interactive chatbot that integrates comprehensive data retrieval, embeddings, and AI-generated responses. By merging web scraping, NLP techniques, and API integration, this solution serves as a powerful tool for efficiently querying extensive documentation.