**Technical Documentation: Natural Language to SQL Query System**

**1. Overview**

This system enables natural language interaction with SQL databases using AI-powered query generation. Users can:

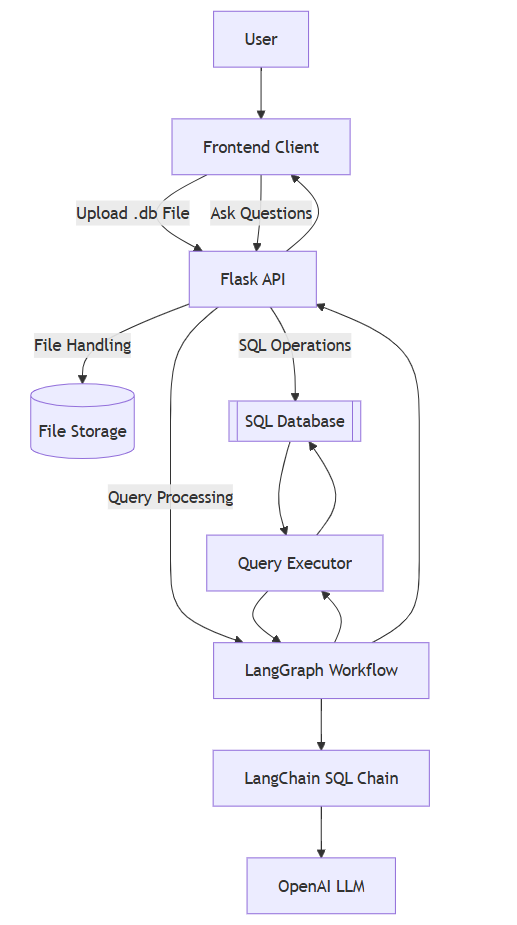
* Upload SQLite database files
* Ask questions in plain English
* Receive generated SQL queries and results
* Maintain contextual conversation history

**Key Technologies**

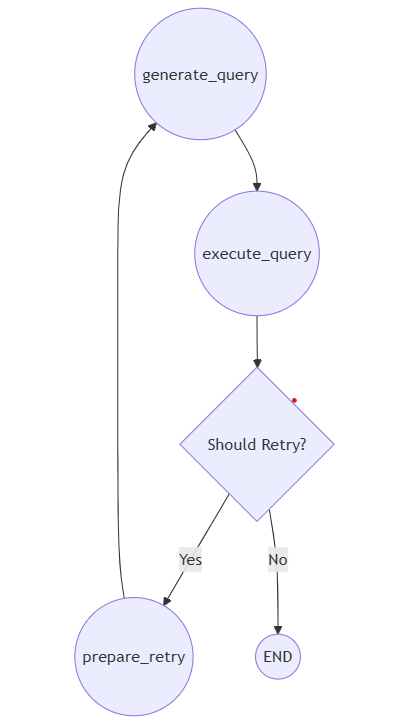
|  |  |
| --- | --- |
| Technology | Purpose |
| Flask | Backend REST API |
| LangChain | SQL query generation framework |
| LangGraph | Workflow management with retry logic |
| OpenAI GPT-4o-mini | LLM for natural language processing |
| SQLAlchemy | Database interaction |

**2. System Architecture**

**2.1 High-Level Architecture**



**2.2 LangGraph Workflow**



**3. Core Components**

**3.1 Backend (Flask API)**

|  |  |
| --- | --- |
| Component | Description |
| File Upload Handler | Validates and stores .db files (max 5MB) |
| Question Endpoint | Processes natural language questions with history context |
| Session Manager | Maintains conversation history (last 3 questions) |
| Error Handler | Manages database errors and API exceptions |

**3.2 AI Processing**

|  |  |
| --- | --- |
| Component | Description |
| History-Aware Chain | Maintains context using FewShotPromptTemplate |
| SQL Query Generator | Creates schema-safe queries with LangChain's create\_sql\_query\_chain |
| Retry Mechanism | Handles SQL errors with 1 automatic retry |
| Result Formatter | Structures results as {columns: [], data: []} JSON |

**3.3 Data Layer**

|  |  |
| --- | --- |
| Component | Description |
| SQL Database | SQLite interface using SQLAlchemy |
| Table Info Fetcher | Retrieves schema information via db.get\_table\_info() |
| Query Executor | Runs generated SQL with parameter sanitization |

**3.4 Frontend (Client-Side)**

|  |  |
| --- | --- |
| Component | Description |
| File Uploader | Drag-and-drop .db file interface |
| Chat Interface | Message-style Q&A display with history |
| Result Visualizer | Tabular display of SQL results |

**4. Workflow Details**

**4.1 File Upload Process**

1. Client uploads .db file via POST /api/upload
2. Server validation

**4.2 Query Processing**

1. Receive question + history via POST /api/ask
2. Generate SQL query using FewShotPromptTemplate
3. Execute query with error handling
4. Return formatted response

**5. API Endpoints**

**5.1 POST /api/upload**

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| file | binary | SQLite database file |

**5.2 POST /api/ask**

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| question | string | Natural language question |
| history | array | Previous questions (max 3) |

**6. Error Handling Mechanism**

1. **Input Validation Errors**
   * + Check file extension against ALLOWED\_EXTENSIONS
2. **Database Connection Errors**
   * + Check global db object initialization
     + Log connection errors for debugging
3. **SQL Query Generation Errors**
   * + Automatic retry mechanism (max 1 retry)
     + Append error to conversation history
     + Use updated context for retry attempt
4. **SQL Execution Errors**
   * + Catch exceptions during db.run()
     + Parse SQLite error messages
5. **Retry Mechanism**
   * + SQL execution error detected
     + Retry count < max retries (1)
     + Error is recoverable (e.g., syntax error)
6. **LLM API Errors**
   * + Retry transient errors (e.g., rate limits)
     + Fallback to simpler query generation

**7. Key Implementation Components**

**1. Custom Prompt Engineering**

* A structured prompt is designed to guide the language model in generating SQL queries.
* The prompt ensures adherence to SQLite syntax, escaping reserved keywords, and applying query limits.
* Uses a combination of historical queries and predefined examples to enhance accuracy.

**2. Few-Shot Learning for SQL Query Generation**

* The system includes predefined examples to improve query generation.
* The FewShotPromptTemplate provides structured input-output pairs, demonstrating the expected SQL query format.
* Example queries include filtering by date, aggregating inflation rates, and retrieving records by region.

**3. Query State Management**

* The QueryState class maintains the following:
  + User question
  + Conversation history
  + Generated SQL query
  + Query execution results
  + Retry count to handle errors effectively

**4. LangGraph-Based Memory and Execution Flow**

* The system uses LangGraph to structure the workflow for query processing.
* Key steps in the workflow:
  1. Generate SQL Query: LLM generates the query using the few-shot learning template.
  2. Execute SQL Query: The query is executed against the SQLite database.
  3. Error Handling and Retry: If the query fails, a retry mechanism adjusts the query.

**5. Context Awareness in Query Execution**

* The model utilizes historical conversation context to improve query accuracy.
* The last few interactions (controlled by HISTORY\_WINDOW\_SIZE) are appended to the prompt.
* This ensures queries remain relevant in multi-turn conversations.

**6. Retry Logic for Query Correction**

* The system automatically retries query generation if an error is detected.
* The error message is appended to the conversation history for better model interpretation.
* A single retry is allowed to prevent excessive looping.
* Common issues handled:
  + Syntax errors
  + Column mismatches
  + Incorrect table references

**8. Deployment**

**8.1 Backend Requirements**

* Python 3.9+
* SQLite 3.32+
* OpenAI API key
* All required Python libraries are listed in requirements.txt. Install them using:

pip install -r requirements.txt

**8.2 Frontend Requirements**

* Node.js 16+
* npm or yarn
* All required frontend dependencies are listed in package.json. Install them using:

npm install

**8.3 Environment Setup**

export OPENAI\_API\_KEY="sk-..."

**8.4 Running the Service**

**Backend**

flask run --host=0.0.0.0 --port=5000

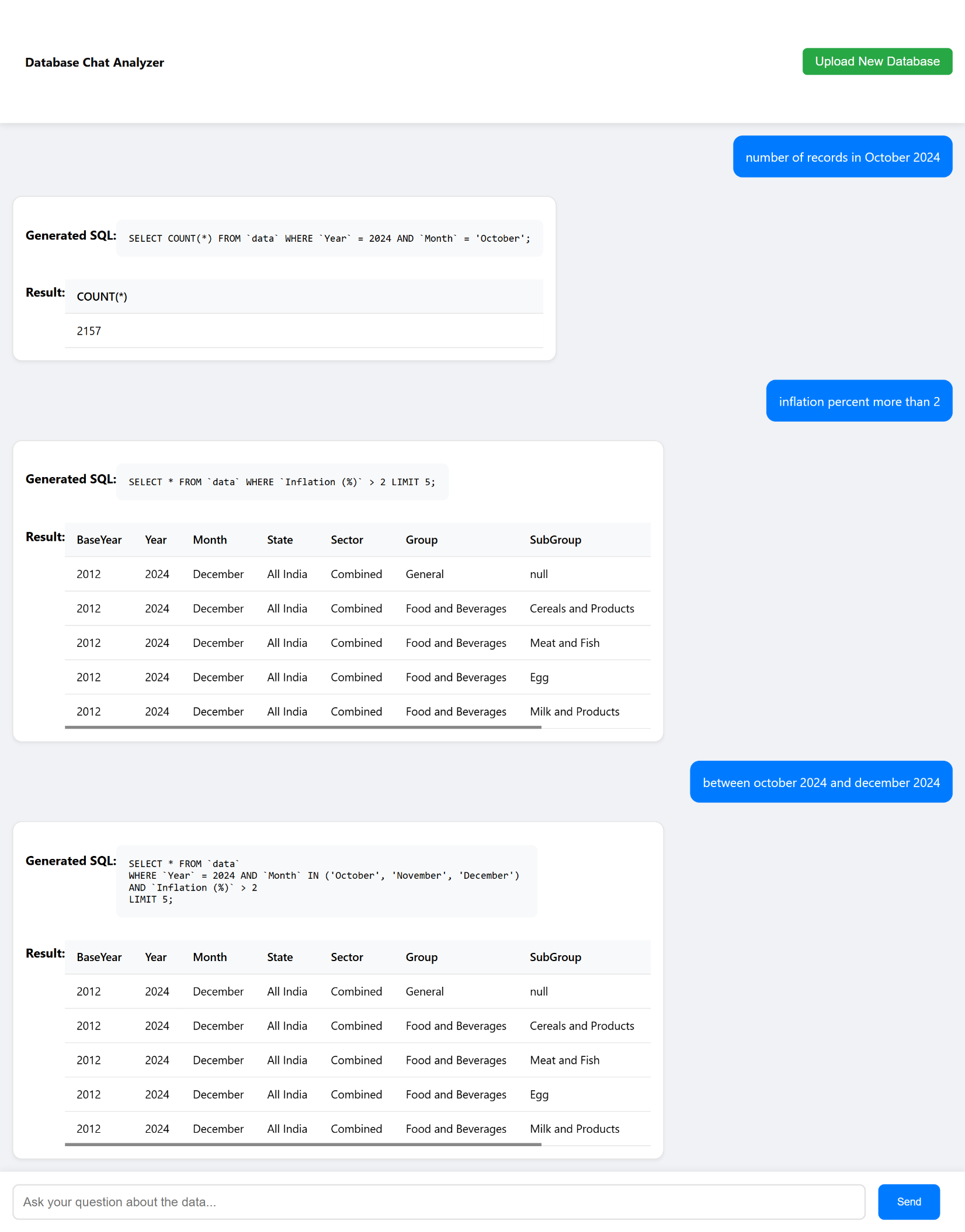
**Frontend**

npm start # or yarn start

**9. Future Improvements**

|  |  |
| --- | --- |
| Category | Proposed Enhancements |
| Security | JWT authentication, Query whitelisting |
| Performance | Query caching, Connection pooling |
| Features | Multi-table joins, Visualizations |
| Scalability | Docker support, Kubernetes deployment |

**10. Screenshot**

****