Documentation: SQL Agent Chatbot for Transaction Data Analysis

**Introduction**

For this project, I developed a specialized chatbot that interacts with structured transaction data stored in a SQLite database. The system was designed to address several key challenges in business data analysis, including natural language query processing, automated reporting, and anomaly detection. This documentation explains my design choices, implementation details, and the overall workflow of the system.

**Choosing SQL Agent Over RAG for Structured Data**

When working with structured data like transaction records, I deliberately chose to implement a SQL Agent approach rather than a Retrieval-Augmented Generation (RAG) system for several important reasons:

Data Structure Alignment: Our transaction data had a well-defined schema with clear tables (transactions, locations, items) and relationships. A SQL Agent can leverage this structure more effectively than RAG.

Precision Requirements: Financial data requires exact matches - we can't afford "similar" transactions when querying specific amounts or dates. SQL provides deterministic results.

Computational Efficiency: For analytical queries (sums, averages, groupings), SQL engines are vastly more efficient than vector similarity searches used in RAG.

Challenge Constraints: The project specifically required working with the provided structured dataset, making the schema-aware SQL approach more appropriate than document-based RAG.

The SQL Agent approach allows users to ask natural language questions which are then converted to precise SQL queries, ensuring accurate results while maintaining an intuitive interface.

**Converting CSV to Database**

The initial dataset was provided in CSV format, which I converted to a SQLite database through the following process:

Schema Design: Analyzed the CSV structure to determine appropriate tables, columns, and data types.

Data Cleaning: Handled missing values, standardized formats, and ensured consistency across records.

Database Creation: Used Python's sqlite3 module to:

Create tables with proper constraints (PRIMARY KEYs, NOT NULL, etc.)

Import CSV data using efficient bulk insert operations

Establish necessary indexes for performance

Schema Documentation: Implemented the get\_database\_schema() function to automatically generate and maintain schema documentation, which is crucial for the SQL generation component.

This conversion process resulted in a jordan\_transactions.db file that serves as the foundation for all subsequent operations.

SQL Generation with OpenAI via Ollama

The core functionality of converting natural language to SQL queries is powered by OpenAI's models through the Ollama interface. Here's how it works in our implementation:

Key Components

Schema Context Injection:

The get\_database\_schema() function retrieves complete schema information

This is dynamically inserted into the LLM prompt to provide context

Prompt Engineering:

python

template = f"""You are a helpful AI assistant that translates natural language queries into SQL code...

Here is the schema of the database:

{{schema}}

Only return the SQL code. Do not provide any explanations...

User query: {{query}}

SQL code:

"""

Clear instructions to focus only on SQL generation

Schema and user query are templated variables

Model Configuration:

Using the openchat model via Ollama (Ollama(model="openchat"))

The LangChain pipeline (prompt\_template | llm) handles the execution

Execution Flow:

User submits natural language query

System retrieves current database schema

LLM generates appropriate SQL query

System executes the query against the database

Results are processed for presentation

Advantages of This Approach

Accuracy: Schema awareness significantly improves SQL generation quality

Maintainability: Automatic schema updates require no manual prompt adjustments

Flexibility: Can handle various query types (SELECT, JOIN, GROUP BY, etc.)

Security: Prevents SQL injection by generating parameterized queries

**System Workflow: Alerts and Reporting**

The system implements several automated workflows for business monitoring:

1. Failure Rate Monitoring

python

def check\_failure\_rates():

"""Checks failure rates against a threshold and alerts managers."""

for location, data in data.items():

if "failure\_rate" in data and data["failure\_rate"] > FAILURE\_THRESHOLD:

send\_email(...)

Runs hourly via schedule.every().hour.do(check\_failure\_rates)

Compares location-specific failure rates against FAILURE\_THRESHOLD (0.05 or 5%)

Sends immediate alerts to location managers when thresholds are exceeded

2. Anomaly Detection

python

def detect\_anomalies():

average\_sales = sum(...) / len(data) if data else 0

for location, data in data.items():

if data["daily\_financial\_report"]["sales"] > average\_sales \* ANOMALY\_THRESHOLD:

send\_email(...)

Runs daily at 10:00 AM

Uses simple statistical method (threshold = 2.0 × average)

Flags unusual sales patterns for investigation

Could be enhanced with more sophisticated ML models

3. Daily Financial Reports

python

def send\_daily\_financial\_reports():

now = datetime.now()

subject\_prefix = f"Daily Financial Report ({now.strftime('%Y-%m-%d')})"

for location, data in data.items():

if "daily\_financial\_report" in data:

send\_email(data["manager\_email"], subject, body)

Delivers standardized reports each morning at 9:00 AM

Includes complete financial snapshots in readable JSON format

Ensures managers start their day with current data

4. Monthly Tax Reporting

python

def send\_monthly\_tax\_collection\_report():

if now.day == 1: # Send on the first day of the month

total\_tax\_collection = sum(...)

send\_email("finance\_manager@example.com", subject, body)

Triggers automatically on the 1st of each month at 8:00 AM

Aggregates tax data across all locations

Sent to central finance team for compliance tracking

**Answer Paraphrasing with DeepSeek API**

To improve user experience, the system uses DeepSeek's API to transform raw database results into natural language responses:

Implementation Details

python

def rephrase\_answer\_deepseek\_api(prompt, data):

deepseek\_prompt = f"""

You are a helpful AI assistant.

The user asked: {prompt}

The data retrieved: {data\_str}

Please rephrase the data into a concise answer.

"""

response = requests.post("https://api.deepseek.com/v1/chat/completions", ...)

return response\_json['choices'][0]['message']['content']

Key Features

Context Preservation: Includes both original query and raw data in the API request

Formatting: Converts database tuples into readable text

Error Handling: Robust exception handling for API failures

Fallback Behavior: Returns simple "No data found" if query returns empty

Benefits

Makes technical database results accessible to non-technical users

Adds explanatory context where helpful

Maintains accuracy while improving presentation

Can highlight key insights from complex result sets

Conclusion

This SQL Agent chatbot system provides a comprehensive solution for interacting with structured transaction data. By combining the precision of SQL with the accessibility of natural language interfaces, it serves both technical and non-technical users effectively. The automated monitoring and reporting features add significant business value by ensuring timely awareness of critical metrics and anomalies. Future enhancements could include more sophisticated anomaly detection algorithms and integration with additional data sources.