Natural Language to Query Generation using Gen-AI

Challenge Overview

Design and develop an AI-powered query system that can accurately translate natural language requests into functional SQL queries for database operations. This system will democratize data access for non-technical staff who need to retrieve financial transaction information but lack SQL expertise.

Problem Statement

In the modern banking environment, financial analysts and customer service representatives frequently need to access transaction data to serve customers and make business decisions. However, many of these professionals lack technical SQL skills required to efficiently query banking databases. This creates bottlenecks in data access, delays, and increases dependency on specialized IT teams.

Your challenge is to create a Generative AI solution that can:

1. Understand natural language descriptions of banking data needs
2. Convert these requests into correct, executable SQL queries for the bank's relational database
3. Return relevant financial transaction information in a user-friendly format
4. Support follow-up questions in a conversational manner to resolve ambiguity resolution.

Technical Requirements

- Develop an AI model that can parse and understand natural language requests about banking transactions, accounts, customers, and branches

- Properly handle sensitive financial data with appropriate security measures

All marked as * are required requirements

Recommended Technology Stack

- Backend: Python 3.10+
- Database: SQLite (required)
- AI/ML: LangChain or LangGraph with OpenAI API or local LLM integration
- Frontend: Python Libraries for Building Interactive Apps (Streamlit/Gradio)
- Package Manager: uv (optional but recommended)
- Testing: Pytest, Unit tests
- Version Control: Git/GitHub

While we recommend certain tools and frameworks, **you are free to use any technology stack** that best suits your team's expertise and approach. Innovation and flexibility are encouraged.

Submission Requirements

- All code must run locally without requiring any third-party infrastructure
- Submissions must include a setup script that initializes the SQLite database with
- sample data
- Include at least 60% of code coverage
- Submit a design/architectural document that includes:
  - Explain your SQL generation approach and why you chose it
  - Describes how your agent architecture works
  - Document about any challenges encountered and how they were addressed
  - Includes performance metrics on your test cases
  - Highlights strengths and limitations of your approach
- Submit your code via a GitHub repository with clear documentation in the main branch.

Database Schema

branch.

Database Schema

Your solution should work with a database that includes tables and data shared via GitHub repository during the hackathon.

Evaluation Criteria

The evaluation process is designed to ensure a fair and comprehensive assessment of each submission. Submissions will be reviewed based on their ability to meet functional requirements, technical implementation quality, user experience, and documentation standards. Bonus points will be awarded for exceptional or innovative features that go beyond the stated requirements. Below is a breakdown of the key areas of evaluation:

Functional Requirements

- SQL Translation Accuracy: Evaluates how accurately the system translates input
- into SQL queries.
- Query Complexity Support: Assesses the ability to handle queries of varying
- complexity.
- Ambiguity Resolution: Measures the system's ability to identify and resolve unclear
- or vague inputs.
- Contextual Understanding: Tests the system's ability to maintain context across

- multiple conversation turns.

Technical Implementation

- Agent Architecture: Focuses on the design, reasoning capabilities, and state
- management of the agent.
- SQL Generation Approach with focus on token minimization: Evaluates the
- efficiency and accuracy of the SQL generation strategy.
- Code Quality & Structure: Reviews the organization, modularity, and adherence to
- design patterns in the code.
- Error Handling & Resilience: Assesses robustness against invalid inputs, SQL
- errors, and edge cases.
- Testing & Quality Assurance: Looks at the extent and quality of automated tests
- and test coverage.
- Security Practices: Ensures protection against vulnerabilities like SQL injection and
- proper data handling.

User Experience & Documentation

- UI Design & Usability: Evaluates the interface for intuitiveness, responsiveness,
- and accessibility.

- Result Presentation: Assesses the clarity and formatting of output, including
- visualizations.
- Documentation & Setup: Reviews of the quality of documentation, including setup
- instructions and inline comments.

- Documentation & Setup: Reviews of the quality of documentation, including setup
- instructions and inline comments.

Bonus Points

- Novel implementation of agent architecture beyond requirements
- Integration with additional tools or APIs that enhance functionality
- Use of uv package manager with proper dependency management
- Exceptional performance on complex edge cases

Automated Evaluation – Critical Reminder

The first level of evaluation is automated. This means:

- The system scans your repository for files in specific folders.
- If files are misplaced (e.g., source code in artifacts/ or demo video in code/), the evaluation may fail.
- Incorrect folder placement can result in your submission being skipped or disqualified, even if the solution is complete and innovative.

Avoid These Common Mistakes

- Committing all files to the root folder.
- Renaming or restructuring the provided template.
- Missing or incomplete README file.
- Uploading video demos for longer than 10 minutes.