Project Overview

Problem Statement

The main problem this prototype addresses is the gap between natural language and structured data. Many people across the healthcare system need quick answers from data, but don't know SQL or how to navigate complex databases. The goal is to build a natural language-to-SQL agent that allows users to ask questions in plain English and get back answers from their data—without needing technical expertise.

Expected Impact

This tool can help a wide range of people across the healthcare system, especially those in operational or support roles who rely on data to make daily decisions:

- **Front desk staff** can ask things like, "What open appointments do we have this week for Dr. Smith?" and get answers instantly.
- **Revenue cycle (RCM) teams** can pull up information such as, "How many claims are still pending by payer/ insurance firm?"
- **Care coordinators** or providers can check, "Which patients haven't completed their annual wellness visit this year?"
- **Schedulers** can quickly ask, "Which departments had the most no-shows last month?"
- **Patient engagement teams** can identify gaps by asking, "How many patients missed their last follow-up call?"

By allowing natural language access to data, this tool supports quicker decision-making, reduces dependency on technical teams, and helps streamline workflows across Commure + Athelas products like Ambient AI (by enhancing real-time understanding), Agents (by powering smarter automations), and RCM (through easier financial insights). It also supports patient engagement and care coordination by making relevant patient data more accessible.

Success Metrics

Success can be measured in a few key ways:

- **Accuracy**: The system returns the correct SQL query and answer for a wide range of natural language questions.
- **Usage**: Number of daily or weekly queries made by non-technical users.
- **Time saved**: Reduction in time spent by operations teams to retrieve or analyze data.
- **Adoption rate**: How widely it's used across different departments (e.g., finance, front desk, care teams).
- **User satisfaction**: Positive feedback from users who find the tool useful, easy to use, and reliable.

Data Design

As part of this project, a custom hospital database schema has been designed and implemented to simulate the structure and operations of a real-world healthcare information system. The schema consists of approximately **18 interrelated tables**, each populated with representative **dummy data** to facilitate development and testing of the agent system.

The database includes core entities typically found in a hospital environment, such as:

- **Patients**: storing personal details, contact information, and risk scores.
- **Doctors** and **Staff**: including their specialties, schedules, and departmental assignments.
- **Appointments**: capturing patient-doctor interactions with timestamps and purposes.
- **Medical Records**: containing diagnoses, treatments, and prescriptions.
- **Billing** and **CPT Codes**: simulating billing procedures and coding for medical services.
- **Medicine**, and **Prescriptions**: tracking medication issuance.
- **Rooms** and **Room Assignments**: managing room availability and allocations.
- **Ambulance** and **Ambulance Log**: simulating emergency transport operations.

To test the natural language processing capabilities of the agent, a **dummy CMS Risk Score** field has been included in the Patients table. This helps evaluate the system's ability to detect and reason with domain-specific data.

Due to the number of tables and the complexity of their relationships, a complete **Entity-Relationship (ER) diagram** has not been included in this report. However, the full schema definition — including all table structures, primary keys, foreign keys, and field attributes — is provided in the Table_Creation.sql file included with the project. This file offers a comprehensive overview of the database design and should serve as a reference for understanding the underlying data model.

Tool/ AI Design

Design Overview

This system uses LangGraph to build a modular, scalable agent workflow for converting natural language questions into reliable SQL queries. LangGraph was chosen over simpler retrieval-augmented generation (RAG) or single-agent approaches because it:

- Handles complex, multi-step reasoning
- Supports multi-agent coordination
- Integrates easily with external tools and APIs
- Provides more control and flexibility for production-grade applications

Prompt & Schema Aware Approach

- The agent is **schema-aware**, meaning it understands the database structure, including tables, fields, and relationships, which helps it generate more accurate queries.
- The prompt includes **strict constraints**: it only allows SELECT queries, avoids destructive operations, returns only the necessary columns, and handles string comparisons in a **case-insensitive** way.
- For more complex queries (like finding max/min values), it includes logic to **handle ties properly** and applies a consistent sort order when needed.
- A separate **critic agent reviews the output**. If there are any issues, the feedback is sent back to the main agent to revise the query, making the system both safe and reliable for operational use.

Agents and Tools in the prototype

Listed in the table are the various agents/tools that have been used in the prototype along with their purpose.

Name	Agent/ Tool	Purpose
Main Agent	Agent	Follow instructions in the prompt. Co-ordinate flow between tools and produce final output.
get_schema	Tool	Parses actual table creation statements to provide full schema context
find_sql_function	Tool	Uses Tavily search to assist the LLM with SQLite-specific syntax
check_error	Tool	Uses Tavily + OpenAI to resolve SQL syntax errors intelligently
validate_sql	Tool	Uses SQLAlchemy to run EXPLAIN and validate the query
update_delete_drop_insert	Tool	Uses regex to block non-SELECT queries for safety
Critic Agent	Agent	Compares NL input, SQL query, and sample output to ensure correctness

Quality Checks

To ensure some level of auditability, a set of test cases has been written covering a range of questions, from simple to more complex queries. The evaluation currently checks whether the generated SQL exactly matches the expected query. If it doesn't, both versions are passed to an LLM judge to determine if they are functionally equivalent. While this gives a basic sense of correctness, there's room to expand this into a more robust evaluation framework. For example, by checking result equivalence, performance, or schema usage patterns.

As for guardrails, these have been introduced both through the prompt (with strict rules around query generation) and through tools that validate and restrict unsafe SQL. However, this is just the starting point. The overall scope for building a fully controlled and auditable agent is much broader.

Sample Agent Flow:

Below is a step-by-step flow for a sample query like "Give me the count of doctors by department":

1. **Agent Invocation**
   The LangGraph agent is triggered with the user's natural language question.
2. **Schema Retrieval**
   The get_schema tool is used to dynamically fetch and parse the database's table structure. This gives the agent accurate, up-to-date knowledge of what tables and fields exist.
3. **Initial SQL Generation**
   Using the schema and prompt context, the agent generates a draft SQL query.
4. **Safety Check (Only SELECT allowed)**
   The generated query is passed through the update_delete_drop_insert tool, which uses regex to ensure that only **safe SELECT queries** can proceed. Any attempts to modify data are blocked.
5. **SQL Validation**
   The validate_sql tool runs the generated query using the EXPLAIN statement via SQLAlchemy. This helps detect structural or syntax issues before running the actual query.
6. **Error Recovery (if needed)**
   If validation fails, the check_error tool is triggered. This uses **Tavily + OpenAI** search to suggest how to fix the query, which is then revised and revalidated.
7. **Query Parsing**
   Once validated, the SQL is parsed out from the model response and prepared for execution.
8. **Critic Agent Review**
   A **critic agent** receives the original question, the SQL query, and the sample output. It checks whether the result actually answers the user's intent.
9. **Refinement or Final Output**
   - If the critic finds issues (e.g., wrong grouping, missing field), it sends the flow back for query revision.
   - If the answer looks correct, the final SQL and results are returned to the user.

Executing the prototype

- To run the prototype, you'll need an OpenAI API key and a Tavily API key (Tavily is free to sign up for).
- Start by using the requirements.txt file to set up your environment and then run the notebook.
- Along with the notebook, there are three .sql files in the SQL_Files/ folder that help create a sample SQLite database. Since it's serverless, setting up the environment will suffice to create the database. Please make sure to replace the path to the .sql files appropriately in the notebook.
- The notebook is organized with clear headings, making it easy to follow along and know exactly where to enter your natural language queries.

At the end of the notebook, you'll find a set of test cases ranging from simple to complex queries. Some of these test cases run perfectly, while others might not be 100% accurate yet. Feel free to use these as examples to try out the agent and get a sense of how it performs, as well as to explore where improvements could be made.

Known Limitations/Next Steps

Privacy Concerns:
Since this system works with sensitive healthcare data, using cloud-hosted models like OpenAI's can raise privacy and compliance issues, especially around HIPAA. To mitigate this, we could explore deploying models locally, using encrypted inputs, or leveraging more privacy-conscious APIs.

Handling Ambiguity in User Queries:
The agent currently lacks the ability to ask follow-up questions when a user query is vague. For example, when asked to "fetch patients at risk," it simply returns the patient with the highest risk score, without knowing what threshold defines "at risk." This could be improved by adding a conversational layer where the agent can ask clarifying questions before attempting to generate SQL.

Misinterpretation of Schema Context:
In some cases, the agent misinterprets how to retrieve certain information. For instance, when asked to return rooms that are occupied, it unnecessarily joins with another table instead of using the status column in the rooms table. This happens because the agent isn't always aware of how each column should be used. A potential fix is to pass more detailed column definitions, including examples or descriptions, as part of the schema input.

Evaluation and Auditability:
The current evaluation process is fairly simple - mainly checking whether the generated SQL matches expected output, and falling back to an LLM judge if not. This can be expanded by comparing result sets, measuring semantic similarity, and integrating evaluation tools like the RAGAS library to better assess how well the agent performs.

Safety and Guardrails:
Basic protections are already in place, for example, tools that block modification queries and restrict access to schema structure. However, this can be made more robust by adding stricter query validation, anomaly detection, query monitoring, and predefined business rules.

Critic Agent Limitations:
The critic currently reviews only the natural language question and the SQL query, without looking at the data returned. Passing a sample of the query output to the critic would make it easier to determine whether the generated query actually answers the user's question.

Schema Token Usage:
The full schema is currently passed to the model on each call, which consumes many tokens and can affect performance. Selectively passing only the relevant parts of the schema - either by inferring table relevance from the question or through schema indexing - could improve both accuracy and efficiency.