LangGraph QA Agent Based On University Database

Design Decisions

1. System Overview

The system translates natural language questions into SQL queries, executes them against a relational university database, and returns a clear, human-readable answer

The execution flow through the LangGraph agent is:

User question → load schema → generate SQL (LLM) → execute SQL → answer (LLM) → Final Answer
. └—— retry (on error) —┘

This flow is implemented using LangGraph to model the agent as a state machine. Each step is represented as a node in the graph, making the system traceable, modular, and easy to debug and understand.

2. Database Design

The database models a university system and includes the following core entities: Teachers, Students, Courses, Course Offerings, and Enrollments, and uses the following relational design principles:

Table	Purpose
Teachers	Teacher entity with required name
Students	Student entity with required name
Courses	Course catalog (unique code, credits > 0 enforced by CHECK constraint)
Course_offerings	A specific course taught by a teacher in a given semester/year
Enrollments	Student enrollment in a course offering, including grade (0-100)

This normalized structure enables complex joins, aggregations (e.g., AVG, COUNT), filtering by semester, teacher, student, or year, and multi-step reasoning queries.

3. LangGraph QA Agent Architecture

LangGraph models the QA process as a stateful graph. Each node has a clear responsibility:

- load_schema – retrieves database schema dynamically
- gen_sql – generates SQL or clarification requests using the LLM
- exec_sql – executes the query safely against the database
- answer – generates a human-readable response

Using LangGraph ensures modularity, clarity, and support for branching logic such as retry loops or clarification paths.

4. DB - Agnostic Design

A key requirement was making the system database-agnostic. Instead of hardcoding table or column names in prompts, the system:

1. Reads the schema dynamically from the database
2. Extracts the *CREATE TABLE* definitions
3. Injects the schema into the LLM prompt

This allows the agent to adapt automatically to schema changes, such as no schema-specific logic is embedded in the agent, and switching databases requires minimal changes.

5. SQL Generation Strategy

The LLM returns strict JSON in one of two formats: {type: 'sql'} or {type: 'clarify'}. This structured JSON makes parsing deterministic and reduces hallucination risks

Only SELECT queries are allowed. Non-SELECT queries are rejected. A LIMIT clause is automatically added when missing to prevent large result sets.

6. Tracing and Observability

Full tracing is implemented internally. Each node records structured events, including timestamps, generated SQL, execution results, and errors (if any). This allows step-by-step debugging and easy identification of failure points

7. Error Handling and Retry Logic

If SQL execution fails, the error message is stored and injected into a retry prompt. The LLM attempts to fix the query up to a configurable maximum number of attempts.
Ambiguous questions - if the question is unclear, the LLM returns *type: "clarify"* instead of executing SQL. If retries are exhausted, the user receives a clarification question.

8. Testing Strategy

The project includes unit tests at three levels:

- Database tests (joins across tables, aggregations, constraints validation)
- SQL generation tests (JSON parsing, LIMIT injection, SELECT-only enforcement, retry prompt validation)
- End-to-end agent tests (path execution, retry recovery, clarification flow, trace validation)

9. Production Considerations

- **Reliability:** add database connection pooling, add LLM retry with exponential backoff, add query timeouts, monitor failure rates.
- **Scalability:** the agent is stateless, which makes it horizontally scalable. To improve performance: cache the schema to reduce DB load, use async LangGraph for higher throughput.
- **Security:** SELECT-only enforcement is already in code; For production, enforce it also at the DB user level to prevert erase of data, use parameterised queries (already

supported via sql_params) to prevent injection, store API keys in environment variables or a secret manager in the cloud.

- **Monitoring:** Emit trace events as structured JSON logs and send to Datadog, metrics for SQL error rate, retry rate, and LLM latency, alerts when error rates exceed thresholds, replace custom tracing with LangSmith.
- **Deployment:** Package the application in Docker image, run pytest in CI/CD on every pull request.