Problem Statement Background

Data lives in databases. Answers live in people's heads as questions.

The gap between the two is SQL — a language that analysts, business users, and even engineers don't always want to write. Simple questions like "how many orders last month?" shouldn't require you to remember table names, figure out joins, or debug syntax errors.

The obvious solution is to throw the question at an LLM with the database schema and ask for SQL. This works... sometimes. It breaks when:

- The schema is large and the LLM hallucinates table names
- The question needs exploration first ("what values exist in this column?")
- The query is wrong and there's no way to catch and fix it
- The question is ambiguous ("show me recent sales" — how recent?)
- The answer requires multiple steps of reasoning
- The LLM runs a `SELECT *` on a 10-million-row table : huge cost

This naive approach hits around 50% accuracy on complex queries. Humans hit 92%.

The gap is where your solution lives.

Problem Statement Objective

Design and build a system that:

- Takes natural language questions as input
- Reasons about the database schema and the question
- Generates and executes safe, efficient SQL
- Returns human-readable answers
- Shows its reasoning (what it did and why)

Ideal system should go beyond simple prompt-to-SQL. It should think, explore, validate, and recover from mistakes.

This is where we want to reach. The more cases you are able to solve, the better.

Sample Queries

Your system should handle questions like these:

Simple

- "How many customers are from Brazil?"
- "List all albums by AC/DC"

Moderate

- "Which 5 artists have the most tracks?"
- "Total revenue by country, sorted highest first"

Requires reasoning

- "Customers who purchased tracks from both Rock and Jazz genres"
- "Which artist has tracks in the most playlists?"

Ambiguous (should clarify or state assumptions)

- "Show me recent orders"
- "Who are our best customers?"

Multi-step / Exploration

- "Are there any genres with no sales?"
- "Which customers have never made a purchase?"

Meta / Introspection

- "What tables exist in this database?"
- "Show me the schema of the Invoice table"
- "Which table has the most rows?"

Sample Workflow

Here's what a thoughtful system might do:

User: "Which customers have never made a purchase?"

System reasoning:

```
├── Need customers with no invoices
├── Checking schema... found Customer table, Invoice table
├── Invoice has CustomerId as foreign key
├── Strategy: LEFT JOIN, filter where InvoiceId is NULL
└── Generating query...
```

Generated SQL:

```
SELECT c.FirstName, c.LastName, c.Email

FROM Customer c

LEFT JOIN Invoice i ON c.CustomerId = i.CustomerId

WHERE i.InvoiceId IS NULL;

Executed. 0 rows returned.
```

Response: "All customers in the database have made at least

one purchase. There are no customers without an invoice."

The system didn't just generate SQL — it explained its approach, handled an empty result gracefully, and gave a meaningful answer.

Database

A sample database (Chinook) will be provided. It's a digital media store with 11 tables — artists, albums, tracks, customers, invoices, employees, etc.

Download: https://github.com/lerocha/chinook-database

We encourage the teams to bring their own dataset and demonstrate additional complexity and edge cases. The more complex scenarios your system handles, the better.

Requirements

Must have:

- Natural language input → SQL → human-readable output
- Works on the provided database
- Demonstrates at least 3 complexity levels (simple, moderate, multi-step)
- Shows reasoning trace (user can see what the system did)
- Read-only queries only (no INSERT, UPDATE, DELETE)
- Handles at least one failure gracefully (error, empty result, or ambiguous input)

Good to have:

- Self-correction (query fails → system retries with a different approach)
- Schema exploration before querying
- Clarifying questions for ambiguous input
- Resource-conscious behavior (no blind `SELECT *`)
- Meta-queries (table info, schema introspection)

Interface:

- CLI is perfectly fine
- Web interface is a bonus, not a requirement

Deliverables

1. Working system with source code

2. Brief documentation explaining your approach

3. Demo showing:

- At least 5 queries of varying complexity
- Where a naive "schema + question → LLM → SQL" approach would fail
- How your system handles it better

4. Short presentation (template will be provided)

Constraints

What's allowed:

- Any LLM (Gemini recommended — GCP free tier is generous)
- Any programming language
- Any libraries or frameworks

What matters:

- You will be evaluated primarily on the complexity and edge cases your system handles
- Teams that build at a lower level of abstraction and demonstrate deeper understanding of the problem will naturally handle more edge cases and score higher
- Be wary of higher levels of abstractions
- Show us what breaks the naive approach and why yours is better

What's NOT allowed:

- Hardcode-ish answers to specific questions
- Using commercial text-to-SQL SaaS products
- Submitting without a working demo

Why This Matters

This is an active problem in industry. Companies like Databricks, Snowflake, and numerous startups are building variations of this — natural language interfaces to data.

Evaluation Focus

- Reasoning depth
  - Does the system think through the problem or just guess?
- Edge case coverage
  - How many complex/ambiguous/multi-step queries does it handle?
- Baseline comparison
  - Can you show where naive approaches fail and yours succeeds?
- Safety
  - Read-only? No runaway queries?
- Transparency
  - Can the user see what happened?
- Robustness
  - What happens when things go wrong?