SQL Agent using LangGraph

Overview

This article describes a table-scoped SQL agent: given a **table name**, its **column metadata**, and a **user question**, it generates **BigQuery SQL**, validates it, executes it, and returns the **most relevant rows** plus a **total record count**.

Table selection (figuring out *which* table to use) is intentionally **out of scope** here and handled by a separate router layer (typically RAG + ranking). This separation keeps responsibilities clean and failure modes diagnosable.

---

1) Problem Statement and Boundaries

1.1 The Problem

Users ask natural-language questions over operational tables: aggregations (“count by status”), drill-downs (“show top issues for X”), definitions (“what is field Y”), and general queries that translate into filters, groupings, and ordering.

1.2 What This Agent Solves

This agent assumes the **correct table is already known** and focuses on producing reliable SQL and results for that single table:

- Plan the SQL steps in plain English (intent → operations).
- Validate filter assumptions against actual table values.
- Generate executable BigQuery SQL.
- Execute safely with a row cap and compute a total count.

1.3 What This Agent Does Not Solve

- **Choosing the right table** from a catalog (RAG/router problem).
- Cross-table joins, multi-dataset reasoning, or schema discovery.

---

2) High-Level Architecture

2.1 Two-Layer System (Conceptual)

- **Layer A: Table Router (Out of scope)**
  Uses metadata embeddings, RAG, heuristics, and confidence thresholds to pick the best table(s).
- **Layer B: Table-Scoped SQL Agent (This article)**
  Converts question → validated plan → SQL → executed results on a known table.

2.2 Inputs and Outputs Contract

Inputs

- `user_query`
- `table_name`
- `table_metadata` (per column: type, description, distinct_count, optional distinct/sample values)
- Two LLMs:
  - `thinking_llm` for planning
  - `base_llm` for generation and validation
- Configuration: `n_iterations` (parallel runs), `record_limit` (default cap)

Outputs

- Generated SQL (`candidate_sql`)
- Rows returned (`raw_results`)
- Total record count (`total_record_count`)
- Plan summary / reasoning (`reasoning`)
- Token usage (input/output)

---

3) Metadata Strategy

3.1 Why Schema-Only Is Not Enough

Column names and types rarely match how users speak. Reliability improves when the model sees:

- Column descriptions (business meaning)
- Distinct counts (cardinality hints)
- Sample/distinct values (grounding filters)

3.2 Metadata Used by the Agent

For each column, the agent can use:

- **Type**: guides filtering/aggregation strategy
- **Distinct count**: helps choose grouping keys and low-cardinality “dimension” fields

- **Distinct values (when small)** and **sample values**: improves filter selection and reduces hallucinated values

3.3 Keyword-to-Column Hints

Before planning, the workflow computes a **keyword → columns** mapping based on literal LIKE matches. This is used as a soft guide to pick relevant columns for filtering or selection.

---

4) Workflow Orchestration with LangGraph

4.1 Why LangGraph Here

This system is not a single prompt. It requires:

- Multiple steps (planning, validating, generating, executing)
- Conditional retries based on observed failures
- Traceable state and histories per node

LangGraph provides a structured, debuggable workflow with explicit nodes and conditional edges.

4.2 State Model

A shared `GraphState` flows through the graph and carries:

- Inputs (query, schema metadata, LLMs)
- Intermediate artifacts (paraphrases, keyword mapping, reasoning, candidate SQL)
- Validation/execution results and feedback
- Histories per stage (reasoner history, generator history, validator history)
- Token accounting

4.3 Graph Nodes (At a Glance)

- `preprocess_query_variation`
- `reasoner`
- `reasoner_validator`
- `sql_generator`
- `context_validator`
- `sql_executor`

4.4 Conditional Routing Logic

The workflow self-corrects via conditional edges:

- If filter validation fails → go back to **reasoner**
- If SQL-context validation fails → go back to **sql_generator**
- If execution returns empty rows → go back to **reasoner** with “relax filters” guidance
- If execution hard-fails → go back to **sql_generator** using error feedback
  A recursion limit is applied to prevent infinite loops.

---

5) Step-by-Step: What Each Stage Does

5.1 Query Preprocessing and Variations

The agent normalises the user query and asks the LLM to produce **five paraphrases**. This increases coverage for ambiguous phrasing and improves the planner’s ability to map intent to the schema.

In parallel, a keyword-presence function builds a **query keyword → columns** mapping from literal matches against the table.

5.2 Reasoner: Plan First, in Plain English

The reasoner produces a short “Plan Summary” describing the SQL stages:

- SELECT fields
- WHERE filters
- GROUP BY + aggregations
- ORDER BY + optional LIMIT (only if semantically required)

This stage keeps a running message history and can incorporate corrective feedback after validation failures or empty results.

5.3 Reasoner Validator: Tool-Verified Filters

The system extracts **string filters** from the plan and validates them using a tool:

- Check if a column contains the proposed value (case-insensitive LIKE)
- Return structured pass/fail with concise feedback

If validation fails, the workflow routes back to the reasoner to adjust columns or values before generating SQL.

5.4 SQL Generator: BigQuery SQL from the Plan

The SQL generator produces a single BigQuery SQL statement based on:

- user question + paraphrases
- validated reasoning plan
- full schema metadata
  It is explicitly instructed to:
- use case-insensitive string matching patterns
- reference only known columns
- avoid LIMIT by default (execution caps the rows anyway)
- include a few additional descriptive columns for context when appropriate

The generator retains a chat history so it can repair SQL using validation or execution error feedback.

5.5 Context Validator: SQL ? Intent Alignment

A separate validator checks the generated SQL against:

- the reasoning plan
- required filters and aggregations
- schema compliance
- case-insensitive filter conventions

It returns strict structured output:

- `valid: true/false`
- `feedback: "" or concise issue description`

5.6 SQL Execution and Result Packaging

The executor:

- strips trailing semicolons
- enforces a **record cap** if the model omitted LIMIT
- computes a **total count** using a COUNT wrapper query that removes top-level ORDER BY/LIMIT to keep BigQuery happy
- marks `empty_results` when the query succeeds but returns zero rows, triggering the “relax filters” loop in the reasoner

---

6) Production Behaviour and Observed Efficacy

6.1 Observed Efficacy

In production usage, the reported efficacy is **~70%+** across varied question types, particularly those that translate cleanly into:

- aggregations and group-bys

- drill-down queries with filters
- “top N” and ordering
- definition/lookup style queries when the data model supports it

6.2 Reliability Patterns That Matter

This agent behaves more reliably than prompt-only SQL generation because it:

- validates filter assumptions against live data
- checks SQL-plan alignment explicitly
- corrects itself via structured retries (not “try again” randomness)
- caps result sizes and returns total counts for UI/UX consistency

6.3 Parallel Runs for Better Coverage

The system can run multiple workflows in parallel (`n_iterations`, default 3). Each run produces a candidate output (SQL + results), which can later be ranked or selected by downstream logic.

Conclusion

This is a production-oriented SQL agent for a known table that:

- turns natural language into an explicit SQL plan
- validates critical filter assumptions with tool checks
- generates BigQuery SQL with schema discipline
- executes safely with row caps and count queries
- self-corrects across planning, generation, and execution failures

The biggest gains typically come from:

- **Type-aware validation** (numbers/dates/ranges validated like strings are today)
- **Candidate ranking** across parallel runs (choose the best result, not just any result)
- **Stronger empty-result recovery** (smarter filter relaxation ordering)
- **Tighter contracts with the table-router layer** (confidence thresholds, fallbacks, and clarifying-question triggers)

Revision #1
Created 2026-01-28 08:54:46 UTC by Soubhik Mazumdar
Updated 2026-01-28 12:25:31 UTC by Soubhik Mazumdar