NL2SQL Agent for Mako Group

Overview

The NL2SQL Agent is a natural-language-to-SQL system built for Mako Group's options market-making desk. Traders ask questions in plain English -- "what was the edge on our quoter trades yesterday?" -- and the agent routes to the correct BigQuery table, generates SQL, validates it, executes it, and returns the answer.

It runs as a **Google ADK sub-agent** behind a LiteLLM proxy, consumed by CLI agents (Claude Code in dev, Gemini CLI in production).

Architecture

```
Trader: "what was PnL by delta bucket yesterday?"
    |
    v
Root Agent (mako_assistant)
    |  Delegates data questions
    v
NL2SQL Sub-Agent (nl2sql_agent)
    |
    |-- 1. vector_search_tables    --> Semantic search: which
table(s)?
    |-- 2. load_yaml_metadata      --> Column names, types,
synonyms, rules
    |-- 3. fetch_few_shot_examples --> Similar validated Q->SQL
pairs
    |-- 4. [LLM generates SQL]
    |-- 5. dry_run_sql             --> Validate syntax, estimate
cost
    |-- 6. execute_sql             --> Run query, return results
    |-- 7. save_validated_query    --> Save for future retrieval
(learning loop)
    |
    v
BigQuery Results --> Formatted answer to trader
```

Agent Structure

| Agent | Role | Model | |-------|------|-------| | **mako_assistant** (root) | Routes questions. Data questions -> nl2sql_agent. General questions -> direct answer. | LiteLLM proxy | | **nl2sql_agent** (sub-agent) | 6-tool chain for SQL generation and execution. Temperature 0.1. | LiteLLM proxy |

Tech Stack

| Component | Technology | |-----------|-----------| | Agent Framework | Google ADK v1.25.1 | | LLM Access | LiteLLM proxy (Claude in dev, Gemini in prod) | | Database | BigQuery (read-only) | | Embeddings | text-embedding-005 via BigQuery ML | | Vector Search | BigQuery VECTOR_SEARCH (COSINE, brute-force) | | Config | pydantic-settings + .env | | Logging | structlog (JSON) | | Container | Docker + docker-compose | | Testing | pytest (155 unit + 20 integration tests) |

Data Model: 13 Tables Across Two Datasets

KPI Dataset (`nl2sql_omx_kpi`) -- Gold Layer

Performance metrics per trade. One table per trade origin, all sharing KPI columns (edge_bps, instant_pnl, delta_bucket, slippage intervals).

| Table | What It Contains | When to Use | |-------|-----------------|-------------| | **markettrade** | Exchange trade KPIs | **DEFAULT** when trade type unspecified | | **quotertrade** | Auto-quoter fill KPIs | "quoter edge", "quoter PnL" | | **brokertrade** | Broker trades (has `account` field) | "BGC vs MGN", broker comparison | | **clicktrade** | Manual click trade KPIs | "manual trade performance" | | **otoswing** | OTO swing trade KPIs | OTO-specific metrics |

Data Dataset (`nl2sql_omx_data`) -- Silver Layer

Raw execution details, timestamps, prices, market data.

| Table | What It Contains | When to Use | |-------|-----------------|-------------| | **theodata** | Theo pricing: vol, delta, vega, theta | "implied vol", "greeks", "fair value" | | **quotertrade** | Raw quoter activity (timestamps, levels) | "what levels at 11:15?" | | **markettrade** | Raw trade execution details | Raw timestamps, fill prices | | **clicktrade** | Raw click execution details | Raw click activity | | **marketdata** | Market price feeds (top-of-book) | "market price", "price feed" | | **marketdepth** | Order book depth, multiple levels | "order book", "bid-ask levels" | | **swingdata** | Raw swing trade data | "swing latency" |

Critical Routing Challenge

`kpi.quotertrade` and `data.quotertrade` are **different tables** with similar names. The agent must disambiguate based on question intent:

- "What was quoter edge today?" -> `kpi.quotertrade` (performance metrics)
- "What levels were we quoting at 11:15?" -> `data.quotertrade` (raw activity)

Same ambiguity exists for `markettrade` and `clicktrade`.

Two-Layer Metadata System

The agent uses two complementary layers to understand what data exists and how to query it.

Layer 1: YAML Catalog (Static, In-Repo)

One YAML file per table stored in `catalog/`. Provides:

- **Column descriptions**: what each column means in trading context
- **Synonyms**: maps trader language to column names (e.g. "edge" -> `edge_bps`)
- **Business rules**: how calculations work, what values mean
- **Routing rules**: which table to use for which question type
- **Disambiguation**: resolves overlapping table names across datasets

```
catalog/
  _routing.yaml              # Cross-dataset routing rules
  kpi/
    _dataset.yaml             # KPI shared columns + routing
patterns
    markettrade.yaml          # 150+ columns with descriptions
    quotertrade.yaml
```

```
      brokertrade.yaml
      clicktrade.yaml
      otoswing.yaml
    data/
      _dataset.yaml             # Data dataset context
      theodata.yaml
      quotertrade.yaml
      markettrade.yaml
      clicktrade.yaml
      marketdata.yaml
      marketdepth.yaml
      swingdata.yaml
```

Example (kpi/markettrade.yaml excerpt):

```yaml
table:
  name: markettrade
  dataset: nl2sql_omx_kpi
  fqn: "{project}.nl2sql_omx_kpi.markettrade"
  partition_field: trade_date
  columns:
    - name: edge_bps
      type: FLOAT64
      description: >
        Edge captured in basis points. Positive = better than fair value.
        Primary performance metric for traders.
      synonyms: [edge, the edge, how much edge, trading edge]
    - name: instant_pnl
      type: FLOAT64
      description: Immediate profit/loss at execution, native currency
      synonyms: [PnL, profit, loss, p&l]
```

The `{project}` placeholder is resolved at runtime from settings, allowing the same catalog to work across dev and prod environments.

Layer 2: BigQuery Vector Embeddings (Dynamic, In BQ)

Three tables in the `nl2sql_metadata` dataset, all using `text-embedding-005` (768-dim vectors):

Table	Rows	Purpose	Used By
schema_embeddings	~17	Dataset and table descriptions	`vector_search_tables`
column_embeddings	~1000	Column names + descriptions + synonyms	Future column routing
query_memory	30+ (growing)	Validated question->SQL pairs	`fetch_few_shot_examples`

Why Two Layers?

Concern	YAML Catalog	Vector Embeddings
Column detail	Full descriptions, types, synonyms	Not needed
Semantic search	Not searchable	COSINE similarity
Maintenance	Edit YAML, commit to git	Auto-populated from YAML + scripts
Learning	Static	`query_memory` grows via save_validated_query
Offline	Works without BQ	Requires live BQ connection

The YAML catalog is the **source of truth** for metadata. The embeddings are a **search index** over that metadata, plus a growing memory of validated queries.

How Embedding and Vector Search Works

Embedding Pipeline

```
1. YAML catalog files ······> populate_embeddings.py ······> BigQuery
tables
   (column descriptions,        (MERGE, idempotent)
(schema_embeddings,
    table descriptions,
column_embeddings,
    example queries)
query_memory)
2. BigQuery tables ····> ML.GENERATE_EMBEDDING ····> embedding column
filled
   (rows with empty             (text-embedding-005,
(768-dim FLOAT64 array)
    embedding arrays)            RETRIEVAL_DOCUMENT)
```

Vector Search at Query Time

```
Trader question: "what was the average edge today?"
         |
         v
    ML.GENERATE_EMBEDDING(question, task_type='RETRIEVAL_QUERY')
         |
         v
    VECTOR_SEARCH(schema_embeddings, COSINE, top_k=5)
         |
         v
    Results: [{table: markettrade, dataset: kpi, distance: 0.12},
...]
```

Key details:

- **RETRIEVAL_DOCUMENT** task type for stored content (when embedding table/column descriptions)
- **RETRIEVAL_QUERY** task type for search queries (when embedding the trader's question)
- **COSINE distance** (lower = more similar)
- **No vector indexes** needed (< 200 rows, brute-force search is fast)
- All operations are **idempotent** (CREATE OR REPLACE, MERGE)

Continuous Learning Loop

When a trader confirms a query result is correct, `save_validated_query` inserts the Q->SQL pair into `query_memory` with an embedding. Next time a similar question comes in, `fetch_few_shot_examples` retrieves it as a reference for the LLM.

```
Trader: "Is this what you were looking for?"
Trader: "Yes"
    |
    v
```

```
save_validated_query(question, sql, tables_used, ...)
    |
    v
INSERT into query_memory + ML.GENERATE_EMBEDDING
    |
    v
Future similar questions find this pair via VECTOR_SEARCH
```

The 6-Tool Chain in Detail

1. vector_search_tables(question)

Finds which BigQuery tables are relevant to a question using semantic similarity against `schema_embeddings`.

Input: Natural language question **Output:** Top-5 tables ranked by relevance with descriptions

2. load_yaml_metadata(table_name, dataset_name)

Loads the full YAML catalog for a specific table -- column names, types, descriptions, synonyms, business rules. Also loads dataset context for KPI/data routing.

Input: Table name + dataset name (for disambiguation) **Output:** Full YAML metadata as string

3. fetch_few_shot_examples(question)

Retrieves similar past validated queries from `query_memory` via vector search. Provides proven SQL patterns for the LLM to follow.

Input: Natural language question **Output:** Up to 5 similar Q->SQL pairs with routing signals

4. dry_run_sql(sql_query)

Validates SQL syntax, column references, table permissions, and estimates cost -- all without executing.

Input: SQL query string **Output:** Valid/invalid status + estimated bytes/MB

5. execute_sql(sql_query)

Executes the validated query. Enforces read-only (SELECT/WITH only), auto-adds LIMIT 1000.

Input: SQL query string **Output:** Row count + result rows as dicts

6. save_validated_query(question, sql_query, ...)

Saves a confirmed-correct Q->SQL pair for future retrieval. Immediately embeds it.

Input: Question, SQL, tables used, complexity, routing signal **Output:** Success/error status

---

Safety Mechanisms

Before-Tool Guard

Every call to `dry_run_sql` or `execute_sql` passes through `before_tool_guard`:

- Extracts the first SQL keyword
- Allows only **SELECT** and **WITH** (CTE)
- Blocks **INSERT, UPDATE, DELETE, DROP, ALTER, CREATE**
- Returns an error dict that the LLM sees as a tool failure

System Prompt Enforcement

The system prompt explicitly instructs:

- NEVER generate INSERT, UPDATE, DELETE, DROP, CREATE, ALTER, or any DDL/DML
- NEVER query tables not listed in the catalog
- NEVER use SELECT * -- always specify columns
- ALWAYS filter on `trade_date` partition column
- ALWAYS use fully-qualified table names
- ALWAYS add LIMIT unless user asks for all rows

Read-Only Architecture

- BigQuery client uses ADC with read-only scopes
- `execute_sql` double-checks first keyword before execution
- No write credentials are configured

Conductor Tracks (Implementation Phases)

Track 01: Foundation -- COMPLETE

Established the project skeleton and infrastructure.

Delivered:

- Repository structure, Dockerfile, docker-compose.yml
- ADK agent skeleton (`root_agent` + `nl2sql_agent`)
- Configuration system (pydantic-settings with `.env`)
- BigQuery client with Protocol-based dependency injection
- Structured JSON logging via structlog
- Dev GCP project setup with sample data

Track 02: Context Layer -- COMPLETE

Built the two-layer metadata system that gives the agent deep knowledge of the data.

Delivered:

- 15 YAML catalog files covering all 13 tables
- Dataset-level routing rules and disambiguation logic
- 30+ validated Q->SQL example pairs across 3 example files
- BigQuery embedding infrastructure (3 tables)
- Embedding pipeline scripts (idempotent, CREATE OR REPLACE / MERGE)
- Vector search validation (5/5 test queries returning correct tables)
- `_routing.yaml` with critical cross-dataset routing patterns

Track 03: Agent Tools -- COMPLETE

Implemented the 6-tool chain that the LLM uses to answer questions.

Delivered:

- `vector_search_tables` -- semantic table routing via VECTOR_SEARCH
- `fetch_few_shot_examples` -- retrieve similar validated queries
- `load_yaml_metadata` -- load YAML catalog for specific tables
- `dry_run_sql` -- validate SQL syntax and permissions
- `execute_sql` -- run query with read-only enforcement
- `save_validated_query` -- continuous learning loop
- Shared dependency injection (`_deps.py`)
- 115 unit tests

Track 04: Agent Logic -- COMPLETE

Wired the tools into the agent with a comprehensive system prompt and safety callbacks.

Delivered:

- Dynamic system prompt (`build_nl2sql_instruction`) with:
  - 7 routing rules, tool usage order, SQL generation rules
  - Date injection, clarification rules, response format
  - Full table catalog embedded in prompt
- `before_tool_guard` callback (blocks DML/DDL)
- `after_tool_log` callback (observability)
- `GenerateContentConfig(temperature=0.1)` for deterministic SQL
- 155 unit tests + 20 integration tests

Track 05: Eval & Hardening -- PLANNED

Will establish accuracy metrics and harden the agent.

Goals:

- Gold-standard evaluation set (50+ questions with expected SQL)
- Accuracy metrics per question type and table
- Retry logic for failed queries (up to 3 attempts)
- Edge case handling (empty results, ambiguous questions)

Track 06: Metadata Enrichment -- PLANNED

Will expand metadata coverage and improve routing accuracy.

Goals:

- Additional example queries per table
- Enriched column descriptions from actual data profiling
- Portfolio-specific routing rules
- Column-level vector search (using `column_embeddings`)

Environment Configuration

Setting	Dev	Production
GCP Project	`melodic-stone-437916-t3`	`cloud-data-n-base-d4b3`
LiteLLM Proxy	`http://localhost:4000`	`https://litellm.production.mako-cloud.com/`
LLM Models	`openai/claude-haiku` / `openai/claude-sonnet`	`openai/gemini-3-flash-preview` / `openai/gemini-3-pro-preview`
Embedding Model	Same project connection	Cross-project: `cloud-ai-d-base-a2df`
Data	Sample/thin slices	Full production data

Switching environments requires only changing the `.env` file -- no code changes.

Running the Agent

```
# Unit tests (default, no live services needed)
pytest tests/
# Integration tests (requires ADC + optionally LiteLLM proxy)
pytest -m integration tests/
# Docker web UI
docker compose up
# Open http://localhost:8000 -> select nl2sql_agent
# Docker terminal mode
docker compose run --rm agent adk run nl2sql_agent
```