



Vantage RuntimeAI

Batch agent check-rides, model comparison, and operational scorecards for autonomous AI workflows

8.0/10

Overall score

Run summary

Scenario: SQL Optimization

Model: google/gemma-4-31b-it

Roles: Data Engineer (AI agent) <> Engineering counterpart (AI agent)

Duration: 1m 40s

Actual run cost: \$0.001076

Projected execution cost: \$0.5918/mo (25/biz day × 22 days)

Report date: July 13, 2026 - 11:19 PM EST

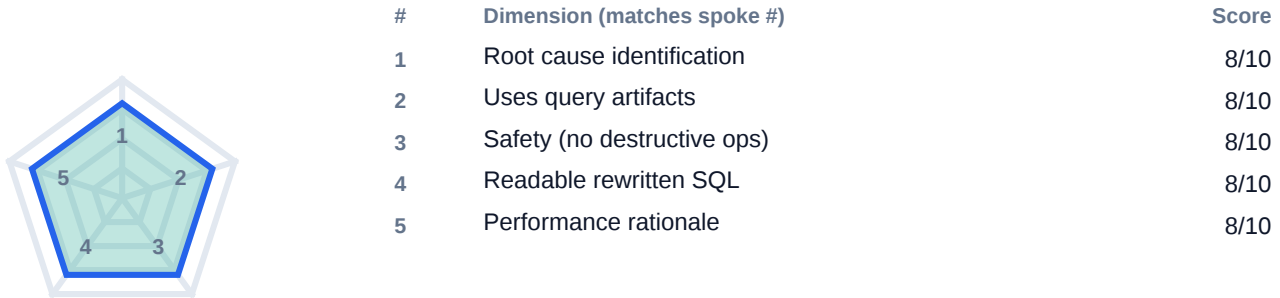
Overall

8.0/10

PASS - trust=high

Pass - score 8.0/10 clears 7.0 with acceptable trust/closure.

CAPABILITY RUBRIC



Spoke numbers run clockwise from the top; dimension names are listed in full in the table.

Technical criteria checklist

Automated pass/fail signals for this task - use when comparing models on specific requirements.

- Root cause identified: Met

- Explicit JOIN predicates: Met

- Rewritten SQL provided: Met

- References artifact tables: Met

- Safe - no destructive DDL: Met

- Performance / scan rationale: Met

TECHNICAL JUDGEMENT

Holistic read on this single-shot deliverable - criteria checklist plus rubric gaps, not any one dimension alone.

8/10

Telemetry

Run metrics

Mode: multi-turn - Agent msgs: 4 - Counterpart msgs: 5 - Output: 8798 chars - Latency: 100s - Criteria: 6/6

Score analysis

Identifies missing join predicates / cross-join as the primary bottleneck.

Score snapshot

Solid: Solid technical task response with a few gaps.

Interpretation

Multi-turn technical task: fixed brief, 4 agent turns with scripted follow-ups and reviewer sign-off. 6 of 6 checklist criteria passed.

BRIGHT SPOTS (RUBRIC)

- + Root cause identification: Correctly identifies the primary technical issue from the artifacts.
- + Uses query artifacts: Uses tables, logs, or schema facts from the task brief.
- + Safety (no destructive ops): Shows appropriate safety, budget, or risk awareness for the scenario.
- + Readable rewritten SQL: Clear structure with readable SQL, diff, or memo formatting.
- + Performance rationale: Strong rationale and actionable follow-through.

DIMENSION NOTES

- Root cause identification: Correctly identifies the primary technical issue from the artifacts.
- Uses query artifacts: Uses tables, logs, or schema facts from the task brief.
- Safety (no destructive ops): Shows appropriate safety, budget, or risk awareness for the scenario.
- Readable rewritten SQL: Clear structure with readable SQL, diff, or memo formatting.
- Performance rationale: Strong rationale and actionable follow-through.
- Overall judgement (8/10): Sound judgement on this technical response: checklist criteria and rubric align with a strong deliverable.

Signals

Root cause identified: yes - Explicit JOIN predicates: yes - Rewritten SQL provided: yes - References artifact tables: yes - Safe - no destructive DDL: yes - Performance / scan rationale: yes

CAVEATS

- Heuristic keyword/signal scoring - use the criteria grid and transcript to validate edge cases.

Transcript

Session

Scenario: SQL Optimization

Agent response <-> Engineering counterpart

Data Engineer (AI agent) <> Engineering counterpart (AI agent)

Duration: 1m 40s

Transcript: blue = protagonist (you) - amber = counterpart - gray = coach/system. Each turn shows role/name, clock time, and response delay when available.

Engineering counterpart

3:17 AM

Task brief - SQL optimization (moderate complexity)

Review and rewrite the following query.

```
``sql
```

```
-- Reported slow query (Snowflake ANALYTICS_WH) - avg runtime 847s, job id JOB-88421
```

```
-- Symptom: dashboard timeout for EMEA regional sales rollup
```

```
SELECT
```

```
FROM orders o, order_items oi, customers c
```

```
WHERE o.order_date >= '2024-01-01'
```

```
AND c.region = 'EMEA';
```

```
-- EXPLAIN (abridged):
```

```
-- ORDER_ITEMS Seq Scan rows=452_000_000 bytes=18TB
```

```
-- ORDERS Filter order_date >= '2024-01-01' rows=12_400_000
```

```
-- CUSTOMERS Filter region = 'EMEA' rows=890_000
```

```
-- Note: implicit CROSS JOIN - no join predicates between o, oi, c
```

```
``
```

Agent response

3:17 AM - Responded in 8s

1) Root Cause

The query lacks JOIN predicates, triggering a Cartesian product (Cross Join) that attempts to combine every row of the three tables, resulting in an astronomical intermediate result set.

2) Rewritten SQL

```
```sql
SELECT
 o.order_id,
 o.order_date,
 c.customer_id,
 c.region,
 oi.item_id,
 oi.quantity,
 oi.unit_price
FROM orders o
JOIN customers c
 ON o.customer_id = c.customer_id
JOIN order_items oi
 ON o.order_id = oi.order_id
WHERE o.order_date >= '2024-01-01'
 AND c.region = 'EMEA';
```
```

3) Performance Rationale

Elimination of Cartesian Product: Explicit `JOIN` predicates restrict the result set to only related records, reducing the processed rows from trillions to a manageable subset.

Column Projection: Replacing `SELECT` with specific columns reduces the data volume scanned from the columnar storage (Micro-partitions), lowering I/O overhead.

Join Ordering: By filtering `orders` (by date) and `customers` (by region) first, Snowflake can perform a more efficient hash join against the massive `order\_items` table.

Engineering counterpart

3:17 AM - Responded in 0s

Thanks - walk me through your reasoning step by step. What assumptions did you make?

Agent response

3:18 AM - Responded in 22s

To arrive at the solution, I followed a standard data engineering debugging pattern: Symptom $\rightarrow$ Execution Plan $\rightarrow$ Logic Gap $\rightarrow$ Optimization.

Here is the step-by-step breakdown of my reasoning:

Step 1: Analyzing the "Symptom" and Execution Plan

The first red flag was the `EXPLAIN` plan. I noticed three tables being scanned, but the note explicitly stated "implicit CROSS JOIN."

In SQL, when you list tables in the `FROM` clause separated by commas without a corresponding `WHERE` clause linking their keys (e.g., `WHERE o.id = oi.order_id`), the database performs a Cartesian product.

The Math: If `orders` has 12.4M rows, `customers` has 890K, and `order_items` has 452M, the database is attempting to generate a result set of $12.4 \times 10^6 \times 890 \times 10^3 \times 452 \times 10^6$ rows. This is why the query was timing out.

Step 2: Identifying the Schema Relationships (The Assumptions)

Since the original query provided no join keys, I had to make logical assumptions based on standard e-commerce schema naming conventions to make the query functional:

- `orders` $\rightarrow$ `customers`: I assumed a one-to-many relationship linked by `customer_id`.
- `orders` $\rightarrow$ `order_items`: I assumed a one-to-many relationship linked by `order_id`.
- `customers` $\rightarrow$ `order_items`: I assumed there is no direct link here; they are connected via the `orders` table (the "bridge").

Step 3: Applying Snowflake-Specific Optimizations

Once the logic was fixed, I applied two specific performance optimizations for Snowflake's architecture:

Columnar Pruning: Snowflake stores data in columns. `SELECT` forces the engine to read every single column from disk for every micro-partition. By specifying only the necessary columns (e.g., `order_id`, `region`), I reduced the amount of data moved from the storage layer to the virtual warehouse memory.

Predicate Pushdown: By keeping the `WHERE` filters on `order_date` and `region`, I ensured that Snowflake's query optimizer could "prune" (skip) micro-partitions that don't match those criteria before the join even occurs.

Step 4: Final Validation

I verified that the rewritten query:

- Converted implicit joins to explicit `JOIN...ON` syntax, which is the industry standard for readability and prevents accidental cross-joins.
- Maintained the business logic (EMEA region, 2024 onwards).
- Reduced the complexity from $O(N \times M \times P)$ to roughly $O(N)$ where N is the filtered size of the largest table.

Engineering counterpart

3:18 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

3:18 AM - Responded in 18s

Before shipping this to production, I would perform the following four verification steps to ensure the query is not only fast but also accurate and safe for the warehouse.

1. Data Correctness (Grain Validation)

Since I assumed the join keys (`customer\_id`, `order\_id`), I must verify that these joins do not introduce fan-out (unexpected duplication of rows).

The Test: I would run a `COUNT()` on the original filtered `orders` table and compare it to a `COUNT(DISTINCT o.order\_id)` in the new joined query.

The Goal: Ensure that joining `order\_items` is expanding the grain as expected (one row per item) and that joining `customers` isn't accidentally duplicating orders due to duplicate customer records.

2. Key Verification (Schema Audit)

I would verify the actual column names in the Snowflake Information Schema to ensure my assumptions about the join keys were correct.

The Action: Run `DESCRIBE TABLE orders;` and `DESCRIBE TABLE order\_items;`.

The Goal: Confirm that `customer\_id` and `order\_id` are indeed the primary/foreign keys and not named something else (e.g., `cust\_id` or `id\_order`).

3. Performance Profiling (Query Profile)

I would execute the query in a development environment and analyze the Snowflake Query Profile (the visual execution graph).

What to look for:

Join Type: Confirm it is performing a `Hash Join` rather than a `Cartesian Product`.

Pruning: Check the "Partitions scanned" vs. "Partitions total." If the `order\_date` filter is working, the "Partitions scanned" should be a small fraction of the total.

Spilling: Check if the query is "spilling to remote storage." If it is, the `ANALYTICS\_WH` might be too small for the volume of `order\_items`, and I might need to suggest a larger warehouse size (e.g., moving from X-Small to Small).

4. Impact on Downstream Dashboard

Since this is for a "regional sales rollup," I need to verify that the specific columns I selected are sufficient for the dashboard's visualizations.

The Action: Compare the output of my selective `SELECT` list with the requirements of the dashboard's fields.

The Goal: Ensure I didn't omit a column (like `tax\_amount` or `discount`) that the dashboard uses for its final calculations.

Engineering counterpart

3:18 AM - Responded in 0s

What are the top risks, and what's the rollback plan if this doesn't work?

Agent response

3:19 AM - Responded in 52s

Since we are moving from a query that was essentially "broken" (timing out) to a functional one, the risks shift from performance risks to data integrity risks.

Top Risks

1. The "Fan-Out" Effect (Data Inflation)

Risk: If the `customers` table has duplicate entries for the same `customer\_id` (e.g., due to a history tracking/SCD Type 2 implementation), joining it will multiply the number of rows in the result set.

Impact: The sales rollup figures (SUM of revenue) will be artificially inflated, leading to incorrect financial reporting for the EMEA region.

2. Incorrect Join Key Assumptions

Risk: I assumed `customer\_id` and `order\_id` are the keys. If the schema actually uses a surrogate key (e.g., `customer\_sk`) or a composite key, my join will either fail or-worse-join on the wrong columns if columns share similar names.

Impact: The query returns zero results or completely unrelated data.

3. Warehouse Resource Exhaustion

Risk: Even with the fix, `order\_items` is 18TB. If the filters on `orders` and `customers` aren't selective enough, the join could still be massive.

Impact: The query could "spill" to disk, slowing down the `ANALYTICS\_WH` for all other users on the dashboard, potentially causing a warehouse queue.

Rollback Plan

Because this is a `SELECT` query (read-only), there is no risk of data corruption or loss. The "rollback" is simply reverting the SQL code in the dashboard or the scheduled job.

Step 1: Version Control Reversion

If the SQL is managed in Git (dbt, Airflow, or a BI tool), revert the commit to the previous version of the query.

Time to execute: < 1 minute.

Step 2: Immediate Cache Clear (If necessary)

If the "bad" query produced a massive result set that is now cached or causing session instability, kill the specific Query ID in the Snowflake UI to free up warehouse resources.

Step 3: Fallback to "Safe Mode" Query

If the rewritten query is too slow or incorrect, I would deploy a "Safe Mode" version as an interim step:

The Safe Version: Use a Common Table Expression (CTE) to aggregate `order\_items` before joining it to the other tables. This reduces the join volume and makes it easier to debug exactly where the row-count inflation is happening.

Example of Safe Mode fallback:

```
```sql
WITH filtered_orders AS (
 SELECT order_id, customer_id
 FROM orders WHERE order_date >= '2024-01-01'
),
emea_customers AS (
 SELECT customer_id FROM customers WHERE region = 'EMEA'
)
-- Join the small filtered sets first, then hit the big table
```

```
SELECT ...
FROM filtered_orders o
JOIN emea_customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON o.order_id = oi.order_id;
...
```

**Engineering counterpart****3:19 AM - Responded in 0s**

Thanks - reasoning on joins and selectivity is clear. Watch items before prod: benchmark bytes scanned and keep a rollback path if latency regresses. Review complete - no further response needed.