



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: sao10k/l3-lunaris-8b

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

Duration: 24s

Actual run cost: \$0.000310

Projected execution cost: \$0.1705/mo (25/biz day × 22 days)

Report date: July 14, 2026 - 3:27 AM EST

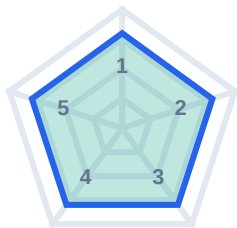
Overall

8.0/10

PASS - trust=high

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

CAPABILITY RUBRIC



#	Dimension (matches spoke #)	Score
1	Root cause identification	8/10
2	Uses query artifacts	8/10
3	Safety (no destructive ops)	8/10
4	Readable rewritten SQL	8/10
5	Performance rationale	8/10

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: 7854 chars - Latency: 24s - 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: 24s

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

Engineering counterpart

7:27 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

7:27 AM - Responded in 4s

Root cause: The query performs an implicit CROSS JOIN on orders, order_items, and customers tables due to missing JOIN conditions, leading to a massive Cartesian product and significant performance issues.

Rewritten SQL with explicit JOIN and selective columns:

```
```sql
```

```
SELECT o.order_id, oi.item_name, c.customer_name, SUM(oi.quantity) AS total_sales
```

```
FROM orders o
```

```
JOIN order_items oi ON o.order_id = oi.order_id
```

```
JOIN customers c ON o.customer_id = c.customer_id
```

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

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

```
GROUP BY o.order_id, oi.item_name, c.customer_name;
```

```
```
```

Performance rationale:

- 1) Explicit JOINS instead of an implicit CROSS JOIN reduce the number of rows being processed significantly.
- 2) Filtering the join conditions early (e.g., order_date and region) further limits the dataset.
- 3) Selecting only necessary columns (order_id, item_name, customer_name, and total_sales) minimizes data transfer and processing.
- 4) The GROUP BY clause introduces additional processing but still provides a more manageable result set.

Engineering counterpart

7:27 AM - Responded in 0s

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

Agent response

7:27 AM - Responded in 6s

Let's break down my thought process and assumptions step by step:

1. Implicit CROSS JOIN identification:

- I recognized that the original query lacked explicit JOIN conditions between the orders, order_items, and customers tables.
- This implied an implicit CROSS JOIN, which would result in the Cartesian product of all rows from each table, explaining the massive row count.

Assumptions:

- The table structures and relationships are as expected (orders has a foreign key to order_items, and both to customers).
- The join is intended to be an inner join, matching rows based on the specified keys.

2. Rewriting with explicit JOINS:

- To resolve the Cartesian product issue, I restructured the query with explicit INNER JOINS, using the primary keys and foreign keys to match relevant rows.
- This allows Snowflake to efficiently join the tables based on the specified conditions.

Assumptions:

- The primary keys (order_id in orders, order_id in order_items, customer_id in customers) are correctly defined and match the foreign key (order_id in order_items referencing orders).
- The data types of the join columns (order_id, customer_id) are compatible for equality comparisons.

3. Selective column retrieval:

- To reduce data transfer and processing, I limited the SELECT clause to only the necessary columns (order_id, item_name, customer_name, and total_sales).
- This assumes that the client application or analytics downstream requirements only need these specific attributes.

Assumptions:

- The client application or analytics tools can effectively utilize the provided columns.
- Additional columns (like order_date) were not required for the specific use case or reporting needs.

4. GROUP BY for summarization:

- To provide useful sales data, I introduced a GROUP BY clause to aggregate the quantity of items per order, item, and customer.
- This assumes that summarizing sales by these dimensions is the primary goal of the query.

Assumptions:

- The total sales (SUM of quantity) is a relevant and meaningful aggregation for the given context.
- Other aggregations (e.g., AVG, COUNT) might be needed depending on the actual use case.

In summary, my approach aimed to address the root cause of the Cartesian product, optimize data retrieval, and provide a suitable result set for the given scenario. The assumptions made were based on the provided query structure, the available information about the tables and relationships, and the intended use of the results.

Engineering counterpart

7:27 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

7:27 AM - Responded in 8s

Before shipping the rewritten query to production, I would take the following steps to verify and refine the changes:

1. Review the EXPLAIN plan:
 - Re-run the EXPLAIN command on the rewritten query to analyze the new execution plan.
 - Compare the plan with the original, looking for improvements in the join order, filtering, and data access.
 - Pay attention to the estimated rows, bytes, and times to gauge the expected performance boost.
2. Test with representative data:
 - Run the query with a sample dataset that mimics the production volume and distribution (if possible).
 - Ensure the query returns the expected results and handles edge cases correctly.
 - Check for any performance regressions or anomalies compared to the original query.
3. Monitor in production (if possible):
 - If possible, run the query against a staging or development environment with similar production traffic.
 - Monitor the query performance, runtime, and resource utilization (e.g., memory, CPU, disk I/O).
 - Compare the metrics to the original query and assess if the improvements are noticeable and sustainable.
4. Consult with stakeholders and review use cases:
 - Discuss the query changes with the client, analysts, or business stakeholders to ensure the rewritten query meets their requirements.
 - Clarify the exact use cases for the results and confirm the chosen columns and aggregations are appropriate.
5. Consider additional indexing and optimization:
 - Review the table structures and index definitions to ensure they support the optimized query.
 - If necessary, create or adjust indexes on join columns, filter conditions, or aggregation keys.
 - Explore other optimization techniques like materialized views, partitioning, or data warehousing patterns if applicable.
6. Schedule a controlled roll-out:
 - Plan a gradual roll-out of the new query to production, starting with a small subset of users or data.
 - Monitor the results and performance closely during the roll-out phase, making adjustments as needed.

By following these verification steps, you can increase confidence in the rewritten query's effectiveness and minimize the risk of performance issues or incorrect results in production.

Engineering counterpart

7:27 AM - Responded in 0s

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

Agent response

7:27 AM - Responded in 6s

Top risks:

1. Performance regression: The rewritten query could potentially lead to slower execution times due to various reasons like inefficient indexing, incorrect join order, or excessive data processing.
2. Data correctness: There's a chance the new query might not produce the exact same results as the original, especially if the rewritten SELECT or JOIN logic differs.
3. Resource overutilization: The optimized query might inadvertently lead to higher resource consumption (e.g., increased memory, CPU, or disk usage) that could impact other critical workloads.

Rollback plan:

1. Quickly revert to the original query: Modify the query to use the original syntax and structure, ensuring it's identical to the production version.
2. Monitor for issues: Closely watch for any adverse effects, such as performance degradation, incorrect results, or resource contention, after reverting the query.
3. Investigate and diagnose: Analyze the changed query, execution plans, and any relevant logs to pinpoint the root cause of the issues and potential missteps.
4. Communicate with stakeholders: Notify the developers, analysts, and business users about the failed optimization attempt, the reasons for reverting, and any necessary corrections or improvements.
5. Re-evaluate and refine: Based on the lessons learned, reassess the original query's performance and look for alternative optimizations or optimization techniques that could be applied safely.
6. Gradual re-attempt: If deemed necessary and after thorough testing, reintroduce the optimization efforts gradually, starting with small-scale testing and monitoring closely for signs of issues before expanding to full production.

The key is to closely monitor the query's performance and results after the change and be prepared to swiftly roll back if problems arise. Thoroughly understanding the query, its impact, and the rollback process ensures a low-risk optimization attempt.

Engineering counterpart

7:27 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.