



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: meta-llama/llama-4-maverick

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

Duration: 15s

Actual run cost: \$0.002320

Projected execution cost: \$1.28/mo (25/biz day × 22 days)

Report date: July 14, 2026 - 3:24 AM EST

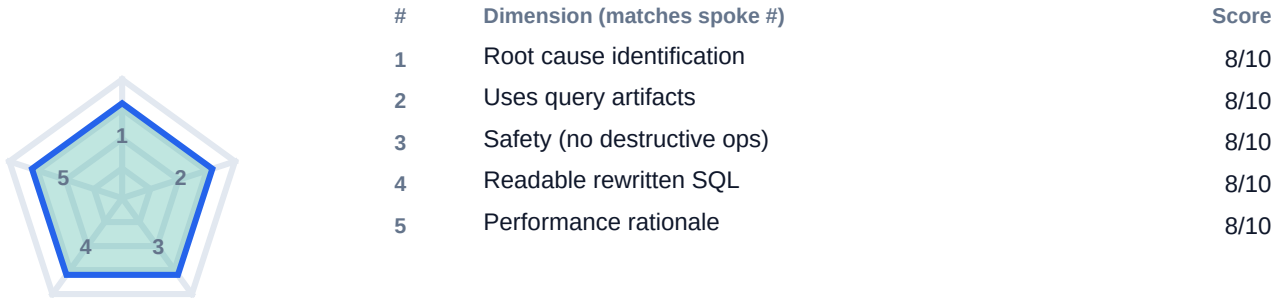
Overall

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: 8302 chars - Latency: 15s - 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: 15s

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

Engineering counterpart

7:24 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

7:24 AM - Responded in 3s

Root Cause

The query is performing an implicit CROSS JOIN between `orders`, `order\_items`, and `customers` due to the lack of explicit JOIN predicates, resulting in a massive intermediate result set that is then filtered. This is causing the query to scan an enormous number of rows (452 million) from `order\_items`.

Rewritten SQL

To optimize the query, we need to add explicit JOIN predicates and select only the necessary columns. Assuming the correct relationships between the tables are `o.order\_id = oi.order\_id` and `o.customer\_id = c.customer\_id`, the rewritten query is:

```
```sql
SELECT
 o.order_id,
 o.order_date,
 c.customer_id,
 c.region,
 oi.item_id,
 oi.quantity
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';
```
```

Performance Rationale

1. Explicit JOIN Predicates: By specifying the JOIN conditions explicitly, we avoid the CROSS JOIN and significantly reduce the number of rows being joined, thus decreasing the computational cost.
2. Selective Columns: Instead of using `SELECT`, we only retrieve the necessary columns, which reduces the amount of data being transferred and processed.
3. Filter Application: The filters on `order\_date` and `region` are still applied, but now they are more effective due to the reduced intermediate result set from the explicit JOINS.

This rewritten query should significantly improve performance by reducing the number of rows being scanned and joined, thus lowering the overall computational cost and memory usage.

Engineering counterpart

7:24 AM - Responded in 0s

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

Agent response

7:24 AM - Responded in 4s

Step-by-Step Reasoning

1. Analyzing the Original Query:

- The query is selecting all columns (`SELECT `) from three tables: `orders`, `order\_items`, and `customers`.
- The query uses an implicit CROSS JOIN (due to the comma-separated table list in the `FROM` clause without any JOIN conditions), which results in a Cartesian product of the three tables.
- Filters are applied on `order\_date` and `region`, but after the CROSS JOIN.

2. Understanding the Problem:

- The EXPLAIN output indicates a sequential scan on `order\_items` with 452 million rows, suggesting a massive intermediate result set.
- The lack of explicit JOIN predicates is the primary cause of the performance issue.

3. Assumptions:

- Relationship between tables: We assumed that `orders` is related to `order\_items` through `order\_id`, and to `customers` through `customer\_id`. This is a common schema design for e-commerce or sales databases.
- Column selection: We selected a subset of columns that might be relevant for an EMEA regional sales rollup, including `order\_id`, `order\_date`, `customer\_id`, `region`, `item\_id`, and `quantity`. The actual columns needed might vary based on the specific requirements of the dashboard or report.

4. Rewriting the Query:

- Explicit JOINS: We rewrote the query using explicit JOINS with the assumed join conditions (`o.order\_id = oi.order\_id` and `o.customer\_id = c.customer\_id`).
- Selective Columns: We replaced `SELECT ` with a list of specific columns that are likely to be required for the sales rollup.

5. Rationale:

- By applying the filters before or during the JOINS (depending on the optimizer's choices), we reduce the number of rows being joined, thus decreasing the computational cost.
- Selecting only the necessary columns reduces data transfer and processing.

Alternative Approaches

- If the assumed join conditions are incorrect, the actual join predicates should be used.
- If additional filters or aggregations are required for the sales rollup, they should be incorporated into the query.

Verification

- To validate the rewritten query, it's essential to verify that the join conditions are correct and that the selected columns meet the requirements of the dashboard or report.
- Running the rewritten query and comparing its results with the original query (if possible) can help ensure that the logic is correct and that performance has improved.

Engineering counterpart

7:24 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

7:24 AM - Responded in 3s

Verification Steps Before Shipping to Production

1. Query Logic:

- Result Set Comparison: Compare the results of the rewritten query with the original query to ensure that the logic is correct and the data is accurate.
- Edge Cases: Verify how the rewritten query handles edge cases, such as NULL values, empty tables, or boundary conditions (e.g., `order\_date` exactly on '2024-01-01').

2. Performance:

- Execution Time: Measure the execution time of the rewritten query and compare it with the original query to ensure that it is significantly faster.
- Resource Utilization: Monitor the resource utilization (CPU, memory, I/O) of the rewritten query to ensure it is within acceptable limits.

3. Data Integrity:

- Data Validation: Validate that the selected columns contain the expected data and that the join conditions are correct.

4. Production Readiness:

- Indexing and Clustering: Verify that the tables are properly indexed and clustered to support the rewritten query.
- Query Optimization: Check that the Snowflake query optimizer is generating an efficient query plan.

5. Testing:

- Unit Testing: Perform unit testing to ensure that the query behaves as expected under different scenarios.
- Integration Testing: Perform integration testing to ensure that the query works correctly within the larger application or dashboard.

6. Documentation and Monitoring:

- Documentation: Update relevant documentation to reflect the changes made to the query.
- Monitoring: Set up monitoring to track the performance and behavior of the rewritten query in production.

By following these verification steps, you can ensure that the rewritten query is correct, performant, and production-ready.

Engineering counterpart

7:24 AM - Responded in 0s

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

Agent response

7:24 AM - Responded in 4s

Top Risks

1. Incorrect Join Conditions:

- If the assumed join conditions (``o.order_id = oi.order_id`` and ``o.customer_id = c.customer_id``) are incorrect, the query will produce incorrect results.

2. Insufficient Column Selection:

- If the selected columns are not sufficient for the dashboard or report, additional columns may need to be added, potentially impacting performance.

3. Performance Regression:

- Although the rewritten query is expected to be faster, there's a risk that it might not perform as well as expected under certain conditions or with changing data volumes.

4. Data Type or Schema Changes:

- If the data types or schema of the underlying tables change, the query may break or behave unexpectedly.

Rollback Plan

1. Revert to Original Query:

- If the rewritten query does not perform as expected or produces incorrect results, revert to the original query (``SELECT FROM orders o, order_items oi, customers c WHERE o.order_date >= '2024-01-01' AND c.region = 'EMEA';``).

2. Temporary Fix:

- If reverting to the original query is not feasible (e.g., due to performance concerns), consider implementing a temporary fix, such as adding additional filters or optimizing the query further.

3. Monitoring and Feedback:

- Continuously monitor the performance and behavior of the query (either the original or the rewritten one) and gather feedback from users to identify any issues or areas for improvement.

4. Iterative Optimization:

- If the rewritten query is not satisfactory, iteratively refine it based on performance metrics, user feedback, and data insights to achieve the desired balance between performance and functionality.

Mitigation Strategies

1. Testing and Validation:

- Thoroughly test and validate the rewritten query before deploying it to production.

2. Gradual Rollout:

- Consider a gradual rollout of the rewritten query to a subset of users or environments to gauge its performance and impact.

3. Version Control:

- Maintain version control of the query (e.g., using a SQL repository or version control system) to facilitate rollbacks and track changes.

By having a rollback plan in place and being prepared to address potential risks, you can minimize the impact of any issues that may arise during or after the deployment of the rewritten query.

Engineering counterpart**7:24 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.