



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-3.3-70b-instruct

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

Duration: 1m 46s

Actual run cost: \$0.001340

Projected execution cost: \$0.7370/mo (25/biz day × 22 days)

Report date: July 14, 2026 - 3:25 AM EST

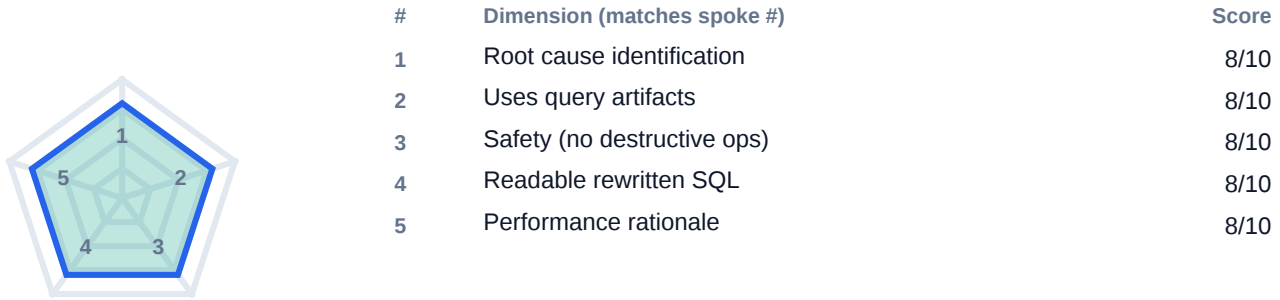
Overall

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: 8829 chars - Latency: 106s - 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 46s

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

Engineering counterpart

7:23 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

7:23 AM - Responded in 8s

Root Cause

The root cause of the slow query is the implicit CROSS JOIN between the `orders`, `order\_items`, and `customers` tables, which results in a massive Cartesian product of rows, leading to a Seq Scan of 452 million rows in the `order\_items` table. Additionally, the `SELECT` statement retrieves all columns, which can increase the amount of data being processed and transferred.

Rewritten SQL

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

Performance Rationale

The rewritten query improves performance by introducing explicit JOIN predicates, which reduces the number of rows being joined and processed. By specifying the join conditions, Snowflake can use more efficient join algorithms and minimize the number of rows being scanned. Additionally, selecting only the required columns (`order\_id`, `order\_date`, `item\_id`, `customer\_id`, and `region`) reduces the amount of data being transferred and processed, further improving query performance.

Engineering counterpart

7:23 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 56s

Here's a step-by-step breakdown of my reasoning:

Step 1: Identify the symptoms and context

I started by reviewing the provided information:

The query is slow, with an average runtime of 847 seconds.

The query is used for a dashboard that displays EMEA regional sales rollup.

The EXPLAIN plan shows a Seq Scan of 452 million rows in the `order\_items` table.

There's an implicit CROSS JOIN between the `orders`, `order\_items`, and `customers` tables.

Step 2: Analyze the implicit CROSS JOIN

I recognized that the implicit CROSS JOIN is likely the primary cause of the query's slowness. A CROSS JOIN combines each row of one table with each row of another table, resulting in a massive Cartesian product of rows. In this case, the CROSS JOIN is between three tables: `orders`, `order\_items`, and `customers`.

Step 3: Determine the correct join relationships

To fix the implicit CROSS JOIN, I assumed the following join relationships based on common database design patterns:

`orders` is joined to `order\_items` on the `order\_id` column, as each order can have multiple order items.

`orders` is joined to `customers` on the `customer\_id` column, as each order is associated with a customer.

These assumptions may not always be correct, and the actual join relationships may vary depending on the specific database schema and requirements.

Step 4: Rewrite the query with explicit JOINS

I rewrote the query using explicit INNER JOINS to specify the join relationships:

```
```sql
FROM
 orders o
 INNER JOIN order_items oi ON o.order_id = oi.order_id
 INNER JOIN customers c ON o.customer_id = c.customer_id
```
```

By specifying the join conditions, Snowflake can use more efficient join algorithms and minimize the number of rows being scanned.

Step 5: Select only required columns

To further improve query performance, I replaced the `SELECT` statement with a `SELECT` statement that only retrieves the required columns:

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

This reduces the amount of data being transferred and processed, which can lead to significant performance improvements.

Assumptions made

The following assumptions were made during the rewriting process:

The join relationships between `orders`, `order\_items`, and `customers` are as described above.

The `customer\_id` column exists in the `orders` table and matches the `customer\_id` column in the `customers` table.

The `order\_id` column exists in both the `orders` and `order\_items` tables and is used to join these tables.

The selected columns (`order\_id`, `order\_date`, `item\_id`, `customer\_id`, and `region`) are sufficient for the dashboard's

requirements.

If these assumptions are incorrect, the rewritten query may need to be adjusted accordingly.

What would you verify before shipping this to production?

Agent response

7:25 AM - Responded in 23s

Before shipping the rewritten query to production, I would verify the following:

1. Join relationships and data integrity

Verify that the assumed join relationships between `orders`, `order\_items`, and `customers` are correct. Check the database schema, documentation, and existing queries to confirm the relationships.

2. Column existence and data types

Ensure that the columns used in the query (`order\_id`, `order\_date`, `item\_id`, `customer\_id`, and `region`) exist in the respective tables and have the expected data types.

3. Query correctness and results

Run the rewritten query on a development or staging environment to verify that it produces the expected results. Compare the results with the original query to ensure that the changes have not introduced any errors or inconsistencies.

4. Performance improvements

Measure the performance of the rewritten query and compare it to the original query. Verify that the changes have resulted in a significant improvement in query execution time.

5. Indexing and optimization

Check if the columns used in the `WHERE` and `JOIN` clauses are properly indexed. Consider adding indexes or modifying existing indexes to further improve query performance.

6. Edge cases and error handling

Test the rewritten query with various input scenarios, including edge cases, to ensure that it handles errors and exceptions correctly.

7. Compatibility with existing applications and workflows

Verify that the rewritten query is compatible with existing applications, reports, and workflows that rely on the original query.

8. Monitoring and logging

Set up monitoring and logging to track the performance and execution of the rewritten query in production. This will help identify any potential issues or areas for further optimization.

Verification steps:

1. Run `EXPLAIN` and `EXPLAIN ANALYZE` on the rewritten query to analyze the execution plan and identify potential bottlenecks.
2. Use Snowflake's built-in query profiling tools to monitor query performance and identify areas for optimization.
3. Test the rewritten query with a representative dataset to ensure it produces the expected results and performs well.
4. Review the query's execution plan and adjust indexing or optimization strategies as needed.

By verifying these aspects, you can ensure that the rewritten query is correct, performs well, and meets the requirements of your production environment.

Engineering counterpart

7:25 AM - Responded in 0s

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

Agent response

7:25 AM - Responded in 18s

Top Risks

1. Data Inconsistency: The rewritten query may produce incorrect or inconsistent results, potentially leading to incorrect business decisions or reporting.
2. Performance Degradation: Despite optimization efforts, the rewritten query may perform worse than the original query, causing increased latency or timeouts.
3. Dependency Breakage: The rewritten query may break dependencies with other applications, reports, or workflows that rely on the original query's output or behavior.
4. Indexing or Optimization Issues: Inadequate indexing or optimization may lead to poor query performance, causing increased resource utilization or timeouts.

Rollback Plan

In case the rewritten query does not work as expected, the following rollback plan can be executed:

1. Revert to the Original Query: Immediately revert to the original query to minimize disruption to business operations.
2. Identify and Analyze the Issue: Investigate the cause of the problem, using tools like Snowflake's query profiling and logging features.
3. Apply Temporary Fixes: Implement temporary fixes or workarounds to mitigate the issue, if possible.
4. Re-Optimize and Re-Test: Re-optimize the query, taking into account the lessons learned from the failed deployment, and re-test it thoroughly.
5. Collaborate with Stakeholders: Work with stakeholders, including business users and developers, to ensure that the revised query meets their requirements and is thoroughly tested.

Rollback Strategy

To minimize the risk of a failed deployment, consider the following rollback strategies:

1. Blue-Green Deployment: Deploy the rewritten query to a separate, isolated environment (e.g., a "blue" environment), and then switch to the new environment once it's verified to be working correctly.
2. A/B Testing: Run the original and rewritten queries in parallel, using A/B testing techniques to compare their performance and results.
3. Canary Releases: Gradually roll out the rewritten query to a small subset of users or environments, monitoring its performance and results before deploying it more widely.

By having a well-planned rollback strategy in place, you can quickly recover from any issues that may arise during the deployment of the rewritten query.

Engineering counterpart

7:25 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.