



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-3.1-8b-instruct

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

Duration: 1m 3s

Actual run cost: \$0.000412

Projected execution cost: \$0.2266/mo (25/biz day × 22 days)

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

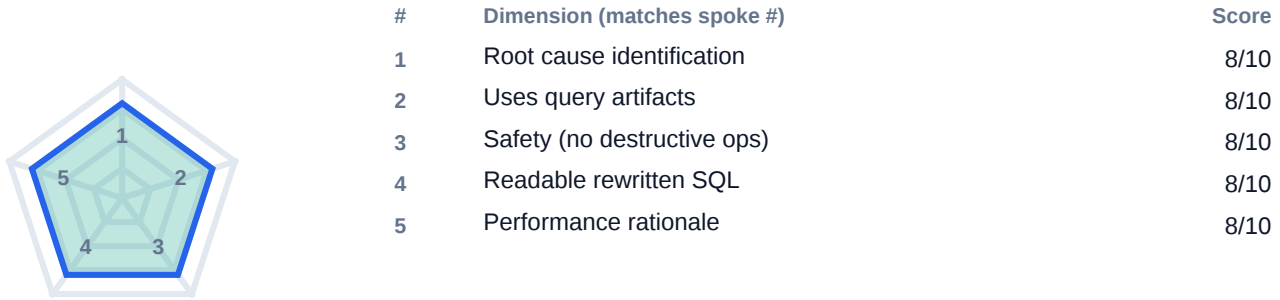
Overall

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: 7700 chars - Latency: 63s - 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 3s

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

Engineering counterpart

3:18 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

3:18 AM - Responded in 5s

Root cause: The query is experiencing performance issues due to an implicit CROSS JOIN with the `orders`, `order\_items`, and `customers` tables, resulting in a large result set and inefficiencies in the join operations.

Rewritten SQL:

```
```sql
```

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

```
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. By adding explicit JOIN predicates (`ON o.order\_id = oi.order\_id` and `ON o.customer\_id = c.customer\_id`), we eliminate the implicit CROSS JOIN and ensure that Snowflake can use optimized join algorithms.
2. Selecting only the necessary columns (`o.order\_id`, `o.order\_date`, `oi.item\_id`, `oi.quantity`, `c.customer\_id`, `c.customer\_name`, `c.region`) reduces the amount of data being transferred and processed, resulting in improved performance.
3. The WHERE clause still filters out irrelevant data (`o.order\_date >= '2024-01-01' and `c.region = 'EMEA'`), ensuring that the query only processes a smaller subset of the data.

Engineering counterpart

3:18 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 8s

Let's break down the reasoning step by step:

Step 1: Understand the problem

I looked at the original query, which has a slow average runtime of 847 seconds.

I noticed that the query uses an implicit CROSS JOIN between the `orders`, `order\_items`, and `customers` tables.

The EXPLAIN output showed that each table is being scanned sequentially, which can be inefficient.

Step 2: Identify the root cause

I determined that the root cause of the performance issue is the implicit CROSS JOIN, which is causing the query to scan each table individually and resulting in large result sets and inefficient join operations.

Step 3: Identify the goal

The goal is to rewrite the query to improve performance while minimizing the changes to the original query.

Step 4: Make assumptions

I assumed that the goal is to improve performance without sacrificing data integrity.

I assumed that the tables have a standard structure, as I made specific changes to the column names (e.g., `o.order\_id` instead of `o.id`).

I assumed that the relationships between the tables are as follows:

An order can have multiple order items (`orders` -> `order\_items`).

An order is associated with one customer (`orders` -> `customers`).

Step 5: Rewrite the query

I added explicit JOIN predicates between the `orders`, `order\_items`, and `customers` tables to eliminate the implicit CROSS JOIN.

I selected only the necessary columns from each table to reduce the amount of data being transferred and processed.

I retained the WHERE clause to filter out irrelevant data, ensuring that the query only processes a smaller subset of the data.

Step 6: Verify the query

The rewritten query should produce the same results as the original query, but with improved performance.

Some potential follow-up steps could include:

Analyzing the query execution plan to ensure that the changes have improved performance.

Optimizing the database schema or indexes to further improve query performance.

Reviewing the query's logic and data integrity to ensure that no critical information is being lost or altered.

Engineering counterpart

3:18 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

3:18 AM - Responded in 6s

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

1. Query results:

Run the rewritten query and compare the results with the original query to ensure that they produce the same output. Verify that the results are accurate and consistent.

2. Performance improvements:

Run the rewritten query and measure its execution time using tools like Snowflake's EXPLAIN or the `TIME` function.

Compare the execution time with the original query's average runtime (847 seconds) to ensure that the rewritten query provides a noticeable performance improvement.

3. Data integrity:

Verify that the query does not introduce any data inconsistencies or logical errors.

Use techniques like data validation or row-level auditing to ensure that the rewritten query does not alter or lose any data.

4. Edge cases:

Test the query with various edge cases, such as:

- Empty tables or missing data.

- Different date ranges or regions.

- Large datasets.

- Concurrent queries or high concurrency.

5. Query plan and execution:

Use Snowflake's EXPLAIN command to analyze the rewritten query's execution plan and ensure that it uses efficient join algorithms and indexes.

Verify that the rewritten query is using Snowflake's optimized join strategies, such as Hash Join or Merge Join.

6. Monitoring:

Set up monitoring and alerting mechanisms to track the rewritten query's performance and detect any regressions.

Regularly review query performance and make adjustments as needed to maintain optimal performance.

By verifying these aspects, you can ensure that the rewritten query is reliable, efficient, and accurate, and that it can handle production workloads without issues.

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 44s

Top risks:

1. Data inconsistencies: If the rewritten query introduces data inconsistencies or alters existing data, it can lead to incorrect results, errors, or even data loss.
2. Performance regressions: The rewritten query might not perform as expected, leading to slow query times, increased latency, or other performance issues.
3. Unintended consequences: The query changes might have unforeseen effects on downstream processes, data pipelines, or other queries, leading to cascading errors or issues.
4. Data loss or corruption: If the query introduces data inconsistencies or alterations, it can lead to data loss or corruption, affecting the integrity of the dataset.

Rollback plan:

To mitigate these risks, I recommend developing a thorough rollback plan that includes:

1. Version control: Ensure that the original query and its changes are version-controlled, so you can easily revert to a previous version if needed.
2. Backup and restore: Regularly back up the database and maintain a restore plan to revert to a previous state in case of issues.
3. Rollback procedure: Define a clear rollback procedure that includes:
 - Reverting to the original query or a previous version.
 - Rolling back changes to the database schema or data.
 - Enabling monitoring and alerting to detect any issues.
4. Contingency planning: Develop a contingency plan to address potential issues, such as:
 - Identifying and isolating the affected components.
 - Assessing the impact on downstream processes or data pipelines.
 - Implementing temporary workarounds or fixes.
5. Communication: Inform stakeholders, including development, QA, and operations teams, about the changes and their potential risks. Establish a clear communication channel for issue reporting and escalation.

Best practices:

1. Gradual deployment: Deploy changes gradually, starting with small subsets of data or regions to test for issues before scaling up.
2. Monitoring and logging: Implement monitoring and logging mechanisms to detect potential issues and track the performance of the rewritten query.
3. Collaboration: Collaborate with the development, QA, and operations teams to ensure a comprehensive understanding of the changes and risks associated with them.

By considering these risks and developing a robust rollback plan, you can minimize the impact of changes and ensure a smooth transition to the rewritten query.

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.