



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-3-12b-it

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

Duration: 46s

Actual run cost: \$0.000510

Projected execution cost: \$0.2805/mo (25/biz day × 22 days)

Report date: July 13, 2026 - 11:17 PM EST

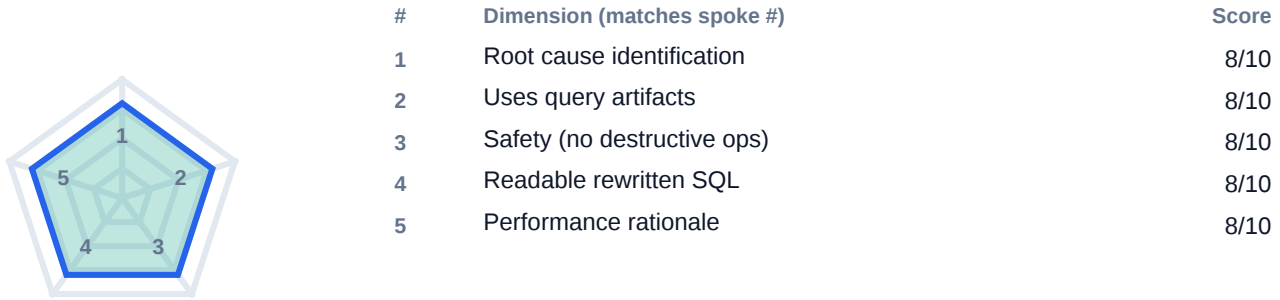
Overall

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: 13571 chars - Latency: 46s - 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: 46s

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

Engineering counterpart

3:16 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

3:16 AM - Responded in 9s

Okay, here's my review and rewrite of the provided query, addressing the performance issues.

1. Root Cause:

The original query utilizes implicit cross joins (Cartesian product) due to the lack of explicit `JOIN` predicates between the `orders`, `order\_items`, and `customers` tables. Snowflake must generate all possible combinations of rows between the tables before filtering, resulting in an extremely large intermediate dataset and slow execution, especially with the large `order\_items` table.

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
 order_items oi ON o.order_id = oi.order_id -- Explicit JOIN predicate
JOIN
 customers c ON o.customer_id = c.customer_id -- Explicit JOIN predicate
WHERE
 o.order_date >= '2024-01-01'
 AND c.region = 'EMEA';
```
```

3. Performance Rationale:

Explicit JOINS: Replacing the implicit cross join with explicit `JOIN` clauses (`JOIN ... ON`) tells Snowflake precisely how the tables are related. This allows the query optimizer to leverage indexes and clustering keys (if defined) and build a much more efficient join strategy (e.g., hash join, sort merge join). The join predicates `o.order\_id = oi.order\_id` and `o.customer\_id = c.customer\_id` are essential for linking the tables correctly.

Selective Columns: Instead of `SELECT \*`, the rewritten query retrieves only the necessary columns (`o.order\_id`, `o.order\_date`, `c.customer\_id`, `c.region`, `oi.item\_id`, `oi.quantity`, `oi.unit\_price`). This reduces the amount of data read from disk and transferred across the network, further improving performance. Selecting only what's needed avoids unnecessary data processing.

Filtering Early: The `WHERE` clause filters data before the join operations, reducing the size of the intermediate datasets. This is critical, ensuring only relevant data participates in the joins.

Potential Clustering/Indexing: This rewrite assumes appropriate clustering keys or indexes exist on `order\_id` in `order\_items` and `orders`, and on `customer\_id` in `customers`. If these don't exist, consider adding them to further accelerate the joins.

By implementing these changes, the query should execute significantly faster, resolving the dashboard timeout issue and improving overall performance.

Engineering counterpart

3:16 AM - Responded in 0s

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

Agent response

3:17 AM - Responded in 11s

Okay, let's break down my reasoning step-by-step, including the assumptions I made along the way.

1. Initial Assessment & Problem Identification (Based on Input)

The Problem Statement: The query was reported as slow (847s average runtime) and causing dashboard timeouts, specifically for an EMEA regional sales rollup. This immediately points to a performance bottleneck.

The `EXPLAIN` Plan: The `EXPLAIN` plan was crucial. The key takeaway was the "Seq Scan" on `ORDER\_ITEMS` scanning 452 million rows and 18TB of data, combined with the "Filter" operations on `ORDERS` and `CUSTOMERS`. The most alarming part was the lack of any explicit join predicates. This screamed "implicit cross join" - a recipe for disaster with large tables.

Understanding Implicit Cross Joins: A cross join (or Cartesian product) creates every possible combination of rows from the involved tables. In this case, it would be creating combinations of every order, every order item, and every customer. That's a massive, unnecessary, and incredibly slow operation, especially when you only want data related to EMEA customers with orders from 2024 onward.

2. Core Reasoning & Rewrite Strategy

The Core Issue: The primary problem was the absence of join conditions. Snowflake had no information about how the tables were related. It just generated all possible row combinations and then filtered.

The Key Insight: We need to explicitly tell Snowflake how the tables are related. The business logic implied clear relationships:

`orders` and `order\_items` are linked by `order\_id`. An order item belongs to a specific order.

`orders` and `customers` are linked by `customer\_id`. An order is placed by a specific customer.

Strategic Rewrite - Explicit Joins: My first step was converting the implicit cross join into explicit `JOIN` clauses. I used `JOIN ... ON` syntax to define these relationships:

`orders o JOIN order\_items oi ON o.order\_id = oi.order\_id` : Links orders to their corresponding order items.

`JOIN customers c ON o.customer\_id = c.customer\_id` : Links orders to the customers who placed them.

Strategic Rewrite - Selective Columns: The second key change was replacing `SELECT` with a list of specific columns. Using `SELECT` retrieves all columns from all tables, even if they're not needed for the rollup calculation. This increases I/O and processing overhead. By specifying only the necessary columns, I reduced the data volume being processed.

3. Assumptions Made

Here are the assumptions I made while formulating this solution. These are critical to understand, as they significantly influence the final performance:

Data Model Accuracy: I assumed the data model accurately reflects the relationships between the tables. That is, `order\_id` and `customer\_id` are indeed the correct foreign keys linking the tables. If the relationships are different, the join conditions would need to be adjusted.

Data Types: I assumed that the data types of the join columns (`order\_id`, `customer\_id`) are compatible for comparison. If there are type mismatches (e.g., one is a string and the other is an integer), Snowflake might perform implicit conversions, which can hurt performance.

Clustering/Indexing (Most Important): This is the biggest assumption. I assumed that appropriate clustering keys or indexes exist on the join columns. Without them, Snowflake will still have to scan the entire tables to find matching rows

Engineering counterpart
3:17 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

3:17 AM - Responded in 13s

Okay, excellent question. Before shipping this rewritten query to production, here's a comprehensive checklist of verification steps I'd take, categorized by importance:

I. Critical Verification (Must-Do)

1. Functional Equivalence Testing (Highest Priority):

Data Comparison: The most important thing is ensuring the rewritten query returns the exact same results as the original query. I'd perform a rigorous data comparison. I'd do this by:

Sampling: Run both queries on a representative sample of data (e.g., a specific date range, a subset of regions) and compare the results row by row.

Aggregation: If the dashboard relies on aggregated metrics (sums, averages, counts), compare the aggregate values produced by both queries.

Edge Cases: Specifically test edge cases:

- Orders with no order items

- Customers with no orders

- Orders from before/after the date range ('2024-01-01')

- Customers from regions other than EMEA (to ensure the filter works correctly)

2. Performance Testing in a Staging Environment:

Realistic Data Volume: Test the query in a staging environment that closely mirrors the production data volume and distribution. Ideally, use data replication to get a near-identical copy.

Baseline Measurement: Measure the runtime of the original query in staging, so you have a clear baseline.

New Query Runtime: Measure the runtime of the rewritten query in staging.

Performance Improvement: Verify that the rewritten query demonstrates a significant improvement in runtime (e.g., a reduction of at least 50% or more, depending on the acceptable performance target).

Resource Consumption: Monitor resource consumption (CPU, memory, I/O) during both query executions. This can reveal potential bottlenecks that might not be obvious from just runtime.

3. Snowflake Query History and Monitoring:

Enable Query History: Ensure that Snowflake query history is enabled and configured to retain sufficient data for analysis.

Dashboard Monitoring: Confirm that the dashboard's performance metrics are being monitored and that any regressions are quickly detected.

II. Important Verification (Highly Recommended)

4. EXPLAIN PLAN Analysis of Rewritten Query:

Join Strategy: Review the `EXPLAIN PLAN` of the rewritten query. Verify that Snowflake is using efficient join strategies (e.g., hash join, sort merge join) and that it's leveraging any available clustering keys or indexes.

Data Scan Patterns: Confirm that the rewritten query is avoiding full table scans wherever possible.

5. Clustering/Index Validation:

Confirm Existence: Verify that the assumed clustering keys or indexes actually exist on the join columns.

Clustering Maintenance: Check the clustering maintenance statistics to ensure that the clustering is still optimal. If the data is changing rapidly, consider increasing the frequency of clustering maintenance.

6. Data Quality Checks:

Null Values: Check for null values in the join columns. Nulls can cause unexpected behavior and potentially degrade performance.

Data Consistency: Ensure data consistency across the tables. Inconsistencies can lead to incorrect results or query failures.

III. Lower Priority Verification (Good to Have)

7. Query Profiling: Use Snowflake's query profiling tools to get granular insights into the query's execution steps. This can help identify specific bottlenecks within the rewritten query.

8. Impact on Dependent Queries:

Identify

Engineering counterpart
3:17 AM - Responded in 0s

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

Agent response

3:17 AM - Responded in 13s

Okay, let's outline the top risks associated with deploying this SQL rewrite and detail the rollback plan.

I. Top Risks

Here's a breakdown of the top risks, categorized by likelihood and impact, along with mitigation strategies.

1. Functional Regression (High Likelihood, High Impact):

Risk: The rewritten query returns incorrect results compared to the original, leading to inaccurate dashboards and potentially flawed business decisions. Even a slight discrepancy can be a major problem.

Mitigation: The rigorous data comparison testing (mentioned in the verification checklist) is the primary mitigation. Extensive sampling, edge case testing, and aggregation comparisons are critical.

2. Performance Degradation (Medium Likelihood, Medium Impact):

Risk: Despite the intention, the rewritten query performs worse than the original. This could be due to unforeseen interactions with Snowflake's query optimizer, inefficient clustering, or issues with the staging environment not perfectly mirroring production.

Mitigation: Thorough performance testing in staging with realistic data volumes. Resource monitoring. Reviewing the `EXPLAIN PLAN` to ensure efficient join strategies.

3. Resource Contention/Impact on Other Workloads (Low Likelihood, Medium Impact):

Risk: The rewritten query consumes excessive resources (CPU, memory, I/O) and negatively impacts the performance of other queries and workloads running in the same Snowflake account.

Mitigation: Resource monitoring during performance testing and production deployment. Consider setting query resource limits to prevent runaway queries. Staged rollout (see rollback plan).

4. Data Type/Compatibility Issues (Low Likelihood, Medium Impact):

Risk: Subtle data type differences or compatibility issues between the tables lead to unexpected query behavior or errors.

Mitigation: Careful review of data types in the join columns and ensuring they are compatible. Explicit type casting might be required in the rewritten query (though I tried to avoid it in the initial version).

5. Dependency Issues (Low Likelihood, Medium Impact):

Risk: Other downstream processes or applications rely on the exact output format of the original query, and the rewritten query's slightly different output format breaks those dependencies.

Mitigation: Thoroughly document all downstream dependencies on the query. Test all affected downstream processes after deployment.

II. Rollback Plan

Here's a phased rollback plan, prioritizing minimal disruption:

Phase 1: Immediate Action (If Critical Errors Detected Immediately After Deployment)

1. Disable the Dashboard: Immediately disable the dashboard the query powers to prevent users from seeing inaccurate data.

2. Rollback the SQL:

Version Control: We must have version control for the SQL code. Revert to the previous version of the query in Snowflake.

Snowflake Time Travel: If applicable and configured, use Snowflake's Time Travel feature to restore the previous version of the query. This is the preferred and fastest method.

3. Verify Rollback: Run the original query and verify that it returns the correct results.

4. Communicate: Communicate the issue and the rollback to stakeholders.

Phase 2: Staged Rollback (If Performance Degradation or Minor Functional Issues)

1. Staged Rollout: If we deployed with a staged rollout (e.g., to a small subset of users first), immediately remove the rewritten query from the test group and revert them to the original query.
2. Monitor Closely: Closely monitor the performance and data accuracy of the original query after the rollback.
3. Root Cause Analysis: Conduct a thorough root

Engineering counterpart**3:17 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.