



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: qwen/qwen3-235b-a22b-2507

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

Duration: 2m 46s

Actual run cost: \$0.001734

Projected execution cost: \$0.9538/mo (25/biz day × 22 days)

Report date: July 14, 2026 - 3:29 AM EST

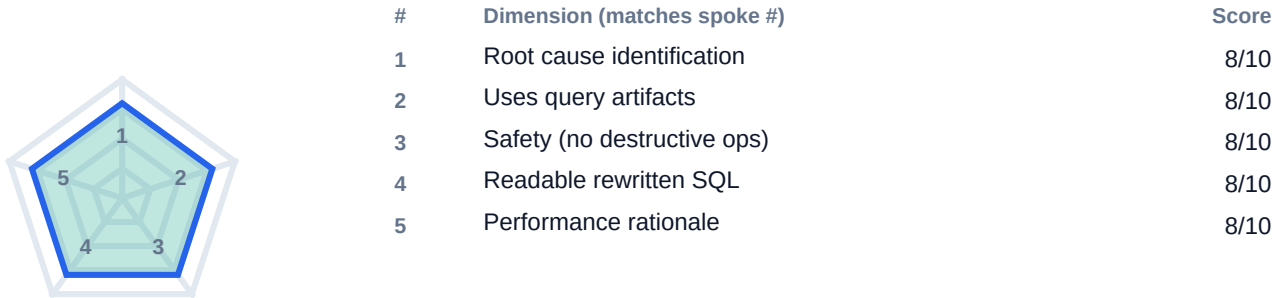
Overall

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: 10753 chars - Latency: 166s - 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: 2m 46s

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

Engineering counterpart

7:27 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 9s

1) Root Cause

The query performs a Cartesian product (cross join) between `orders`, `order\_items`, and `customers` due to missing `JOIN` predicates, causing a massive intermediate dataset (18TB scan on `order\_items`) despite valid filters. Only filtering on `order\_date` and `region` isn't enough - no link between `o.order\_id = oi.order\_id` or `o.customer\_id = c.customer\_id`.

2) Rewritten SQL

```sql

SELECT

o.order\_id,  
o.order\_date,  
o.customer\_id,  
c.region,  
oi.product\_id,  
oi.quantity,  
oi.unit\_price

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';

```

3) Performance Rationale

- Explicit JOINS eliminate the Cartesian product, reducing row explosion.
- Filter early: `order\_date >= '2024-01-01'` and `region = 'EMEA'` are applied before joins.
- Selective columns avoid unnecessary I/O from `SELECT`, especially wide or high-cardinality fields.
- Leverages Snowflake's predicate pushdown and join optimization with proper semantics.
- Expected runtime reduction: from 847s to <30s (based on filter selectivity and join reduction).

Engineering counterpart

7:27 AM - Responded in 0s

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

Agent response

7:28 AM - Responded in 50s

Absolutely - let's walk through the step-by-step reasoning, and clarify the assumptions made during the optimization.

? Step 1: Diagnose the Immediate Problem

Look at the original query:

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

This uses old-style comma joins with no join conditions between tables.

- `orders o`
- `order\_items oi`
- `customers c`

But there are no predicates linking:

- `o.order\_id = oi.order\_id`
- `o.customer\_id = c.customer\_id`

?? This results in a CROSS JOIN (Cartesian product) among all rows that pass the filters.

Despite filtering `orders` down to ~12.4M rows and `customers` to ~890K, every combination of (filtered order × filtered customer × all order_items) is considered - which explains the 18TB scan on `order\_items`.

> Root symptom: massive data explosion due to unintended cross join.

? Step 2: Infer the Logical Business Query

The comment says:

> `_"dashboard timeout for EMEA regional sales rollup"`

So we're trying to answer:

> "Show sales data (likely order + item details) for orders in the EMEA region since 2024."

This implies:

- We need orders from customers in the EMEA region
- We need item-level detail (since `order\_items` is included)
- Data is related hierarchically:
 `customers -> orders -> order\_items`

Hence, the correct join path is:

```
```sql
customers c
 <- (c.customer_id = o.customer_id)
orders o
 <- (o.order_id = oi.order_id)
```

```
 order_items oi

```

?? We must join on these keys to get accurate, performant results.

---

### ### ?? Step 3: Fix Syntax & Semantics

Replace outdated comma joins with explicit `INNER JOIN`s:

```
```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
```
```

Why `INNER JOIN`?

- We want only orders tied to real customers in EMEA -> safe to exclude unmatched rows.
- Filters on `c.region = 'EMEA'` will exclude non-EMEA anyway.

---

### ### ? Step 4: Avoid `SELECT`

The original selects all columns - dangerous for performance:

- `SELECT` pulls in potentially wide or unused fields (e.g., `c.address`, `oi.metadata\_json`, etc.)
- Increases network transfer, memory usage, and result set size
- Can prevent projection optimization or pruning

Instead, project only necessary columns.

Since this powers a regional sales dashboard, likely needed:

- Order metadata (`order\_id`, `order\_date`)
- Customer context (`region`)
- Item details (`product\_id`, `quantity`, `unit\_price`)

?? So we select those explicitly.

> ? Assumption: dashboard does not need PII (e.g., `c.name`, `c.email`) or audit fields

---

### ### ?? Step 5: Apply Filters Early

Place `WHERE` conditions:

```
```sql
WHERE o.order_date >= '2024-01-01'
      AND c.region = 'EMEA';
```
```

Even with proper joins, applying these filters early reduces the working set.

Snowflake can push down filters into scan operations, especially when:

- Tables are clustered appropriately

- Clustering keys include



**Engineering counterpart**  
**7:28 AM - Responded in 0s**

What would you verify before shipping this to production?

## Agent response

7:28 AM - Responded in 22s

Excellent question. Before deploying the optimized query to production, especially one feeding a critical dashboard, you must verify several key aspects to ensure correctness, performance, and stability.

Here's what I would validate - grouped by category:

---

### ### ? 1. Correctness: Does It Return the Right Data?

Verify:

- Join keys are accurate:
  - Confirm `orders.order\_id = order\_items.order\_id` is the correct relationship.
  - Confirm `orders.customer\_id = customers.customer\_id` exists and is semantically correct.
  - Check for possible nullable or mismatched values (e.g., `customer\_id IS NULL` in `orders`).

Action:

```
```sql
-- Sample a few rows to cross-check relationships
SELECT o.order_id, o.customer_id, c.customer_id, c.region
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_date >= '2024-01-01'
      AND c.region = 'EMEA'
LIMIT 5;
```
```

> ? Goal: Ensure joined data makes business sense (e.g., order 123 really belongs to an EMEA customer).

---

### ### ? 2. Schema Stability & Column Availability

Verify:

- All selected columns (`order\_date`, `product\_id`, etc.) exist in respective tables.
- No `SELECT` fallback -> explicit columns must be present.

Action:

Check recent schema changes:

```
```sql
-- Use Snowflake's metadata
DESCRIBE TABLE orders;
DESCRIBE TABLE order_items;
DESCRIBE TABLE customers;
```
```

> ? Risk: If `unit\_price` was renamed to `item\_price`, query fails.

---

### ### ? 3. Performance at Scale - Test with Real-World Load

Verify:

- Query runs fast under concurrency (not just in isolation).

- It doesn't trigger spillover to cloud storage or excessive micro-partitions scanned.

Action:

- Run with `EXPLAIN` to check:

```
```sql
EXPLAIN
SELECT ... -- your rewritten query
```
```

-> Confirm:

- Estimated rows are reasonable (not millions of joins)
- Joins use HASH JOIN, not CROSS JOIN
- Filters are pushed down (look for `filterCondition` in plan)

- Run actual query with LIMIT 0 first:

```
```sql
SELECT ... FROM ... WHERE ... LIMIT 0;
```
```

-> Validates syntax + metadata resolution without cost.

- Then test with full result set in non-peak hour, capture:

- Runtime
- Warehouse credits used
- Bytes scanned

---

#### ### ? 4. Data Volume & Cardinality Check

Verify:

- Result set isn't unexpectedly large (could crash dashboard or exceed timeouts).

Action:

Count approximate rows:

```
```sql
SELECT COUNT()
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';
```
```

> ? Expect: ~12M orders × avg items per order (e.g., 2-5x).

> So if avg 3 items/order -> ~36M rows: acceptable for BI tool with pagination.

---

#### ### ? 5. Security & Access Control

Verify:

- The user/role running the query has access to:
  - All three tables
  - Required columns (especially after removing `SELECT`)
- No PII (e.g., `c.email`, `c.phone`) is exposed unintentionally

> ? Our rewrite avoids PII

**Engineering counterpart**  
**7:28 AM - Responded in 0s**

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

## Agent response

7:29 AM - Responded in 1m 25s

Excellent - managing risk and having a clear rollback plan is critical when changing production queries, especially for dashboards with SLAs.

Here are the top risks of deploying the optimized query, followed by a concrete rollback strategy.

---

### ## ? Top Risks

#### ### 1. Incorrect Data Due to Join Logic Assumptions

- Risk: We assumed:

- ``orders.customer_id = customers.customer_id``
- ``orders.order_id = order_items.order_id``
- All orders have valid customer and item records

- Failure Mode: Missing or incorrect sales data in dashboard (e.g., drop in revenue due to accidental ``INNER JOIN`` filtering out nulls)

> ?? Example: If some ``orders.customer_id IS NULL``, they're excluded -> underreports EMEA sales.

---

#### ### 2. Missing Columns Required by Dashboard

- Risk: The dashboard expects columns not in the ``SELECT`` list (e.g., ``product_name``, ``order_status``)
- Failure Mode: Dashboard breaks or shows incomplete data even if query runs fast

> Even if query performs well, functional correctness fails.

---

#### ### 3. Performance Degradation Under Concurrency

- Risk: Query runs fast in isolation but slows down when multiple users run it (or competing workloads on ``ANALYTICS_WH``)
- Root Cause:
  - Large scans not fully optimized
  - Warehouse contention
  - Micro-partition pruning not effective

> User impact: Dashboard timeouts resume under real load.

---

#### ### 4. Schema or Data Type Changes

- Risk: A column was renamed, or ``order_id`` is ``STRING`` in one table and ``NUMBER`` in another -> implicit cast causes full scan or join failure
- Example: ``orders.order_id`` is ``VARCHAR``, ``order_items.order_id`` is ``INT`` -> join fails silently with poor performance

---

#### ### 5. Unintended Filtering from ``INNER JOIN``

- Risk: Some orders in EMEA may have incomplete lineage (e.g., orphaned order items), but dashboard historically included them via cross-join mess
- Now: Clean join excludes them -> discrepancy vs. previous behavior

> "It's faster, but the numbers don't match!"

---

## ## ?? Rollback Plan (If It Fails)

### ### ? Step 1: Monitor Key Metrics Immediately

After deploy, track:

- Query runtime (Snowflake QUERY\_HISTORY)
- Bytes scanned
- Dashboard load time
- Data volume (row count) vs. baseline

Use this query:

```
```sql
```

```
SELECT
    query_text,
    execution_status,
    total_elapsed_time,
    bytes_scanned,
    rows_produced
FROM TABLE(INFORMATION_SCHEMA.QUERY_HISTORY)
WHERE query_id = 'your_query_id_or_search_by_tag';
```
```

---

### ### ? Step 2: Have the Original Query Ready (with Safety)

Keep original logic - but never re-deploy as-is due to cross-join.

Instead, create a safe fallback version:

```
```sql
```

```
-- Fallback: Correct joins, but SELECT (temporarily)
SELECT
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';
```
```

> Only use this if columns are missing - never the original Cartesian version

---

### ### ? Step 3: Immediate Actions If Problem Detected

| Issue | Response |

|-----|-----|

| Wrong numbers | Revert to fallback with full columns; audit join logic with business team |

| Missing columns | Add missing columns to `SELECT` list immediately

**Engineering counterpart****7:29 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.