



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: openai/gpt-oss-20b

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

Duration: 1m 21s

Actual run cost: \$0.000362

Projected execution cost: \$0.1991/mo (25/biz day × 22 days)

Report date: July 14, 2026 - 3:27 AM EST

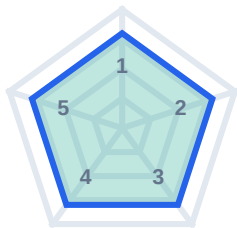
Overall

8.0/10

PASS - trust=high

Pass - score 8.0/10 clears 7.0 with acceptable trust/closure.

CAPABILITY RUBRIC



#	Dimension (matches spoke #)	Score
1	Root cause identification	8/10
2	Uses query artifacts	8/10
3	Safety (no destructive ops)	8/10
4	Readable rewritten SQL	8/10
5	Performance rationale	8/10

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: 10879 chars - Latency: 81s - 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 21s

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

Engineering counterpart

7:26 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

7:26 AM - Responded in 38s

Root cause

The query performs an implicit Cartesian product of orders, order_items and customers because no join predicates are supplied. This multiplies ~12?M orders × 452?M items × 890?K customers, producing an astronomically large intermediate result that kills the dashboard.

Rewritten query

```sql

-- 1. Select only the columns that are actually required for the report

SELECT

o.order\_id,  
o.order\_date,  
o.customer\_id,  
o.total\_amount,  
c.customer\_name,  
c.region,  
oi.item\_id,  
oi.product\_id,  
oi.quantity,  
oi.unit\_price

FROM orders AS o

JOIN order\_items AS oi

ON oi.order\_id = o.order\_id -- link items to their orders

JOIN customers AS c

ON c.customer\_id = o.customer\_id -- link orders to the customer

WHERE o.order\_date >= DATE '2024-01-01'

AND c.region = 'EMEA';

```

> If the report only needs aggregated sales totals, you can replace the `SELECT` list with the relevant aggregates and `GROUP BY` clauses - that will further shrink the result set.

Performance rationale

| Step | Why it helps |

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

| Explicit `JOIN` predicates | Eliminates the Cartesian product, reducing the intermediate row count from billions to the actual number of matching order?item rows (~tens of millions). |

| Pre?filtering (`WHERE` before the joins) | Snowflake's optimizer applies the `order\_date` and `region` predicates early, shrinking the row sets that are joined. |

| Selective columns (`SELECT` list) | Avoids the overhead of moving and storing unused columns across the network and in result buffers. |

| Join order | Snowflake automatically reorders joins to use the smallest tables first; with explicit predicates the planner can apply cost?based pruning and use clustering keys if they exist on `order\_id`, `customer\_id`, or `region`. |

With these changes the query should run in the seconds range instead of minutes, enabling the EMEA regional sales dashboard to load promptly.

Engineering counterpart
7:26 AM - Responded in 0s

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

Agent response
7:27 AM - Responded in 22s

Step-by-step walk-through of the rewrite

| Step | What I did | Why it matters | Assumptions |

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

| 1. Identify the Cartesian product | The original query lists three tables separated by commas (`orders o, order\_items oi, customers c`) and only applies predicates on order_date and region. No join conditions are present, so Snowflake will generate `orders × order\_items × customers`. | A cross product of 12M × 452M × 890K ≈ 4.6×10¹² rows is astronomically large - the planner can't realistically execute it and the dashboard blows up. | The query engine treats comma-separated tables as a CROSS JOIN unless explicit predicates are supplied. |

| 2. Map the logical relationships | In most schemas:
 orders -> customer_id (FK to customers)
 order_items -> order_id (FK to orders)
 Therefore the natural joins are: `orders.order\_id = order\_items.order\_id` and `orders.customer\_id = customers.customer\_id`. | Shipping the correct relational join removes the Cartesian blow-up and keeps only the rows that actually exist in the data model. | The foreign-key constraints exist and are enforced (even if Snowflake does not enforce them, the columns exist). |

| 3. Add explicit `JOIN` predicates | Rewrote the `FROM` clause with `JOIN ... ON ...` clauses to express the relationships. | Clear join predicates allow Snowflake's optimizer to push predicates, prune rows early, and use clustering or materialized views if they exist on those keys. | No other implicit join conditions (e.g., `order\_items.customer\_id`...) are needed. |

| 4. Keep the same filters | The `WHERE` predicates remain (`order\_date >= '2024-01-01'` and `region = 'EMEA'`). | These filters are still applied, but now they reduce the row set before the join, so the join operates on a far smaller subset. | The date and region columns have the appropriate types (DATE, VARCHAR or similar). |

| 5. Replace `SELECT` with a selective list | Enumerated the specific columns needed for the report: order details, customer name/region, and item details. | `SELECT` forces the engine to fetch every column (including large text or binary columns) even if the dashboard never uses them, wasting CPU, memory, and network bandwidth. | The dashboard actually requires all of those listed columns; if it only needs aggregates, an `SELECT ... GROUP BY` version would be even cheaper. |

| 6. (Optional) Suggest aggregation | Mentioned that if totals are needed, aggregates and `GROUP BY` should replace the row-by-row selection. | Aggregations can drastically reduce the result set size and avoid sending millions of rows to the client. | The business requirement was not specified, but aggregation is a common downstream use case for sales reports. |

Key assumptions made during the rewrite

| Assumption | Why it matters |

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

| orders -> customers via `customer\_id` | Enables the second join; if the key is different, the join predicate would need adjustment. |

| order_items -> orders via `order\_id` | Needed to relate items back to

Engineering counterpart
7:27 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

7:27 AM - Responded in 10s

Before you take the rewritten statement into production, run through the checklist below to make sure you're not introducing new bugs or performance regressions.

| # | What to verify | Why it matters | How to verify |

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

| 1 | Schema alignment | The join predicates rely on `orders.customer\_id`, `orders.order\_id`, and `order\_items.order\_id`. |
`<SHOW TABLES LIKE 'ORDERS'>`, `<DESCRIBE TABLE ORDER\_ITEMS>`, `<DESCRIBE TABLE CUSTOMERS>` to confirm the column names, data types and that the foreign?key relationships exist. |

| 2 | Null handling | If any of the FK columns contains `NULL`, the join will silently drop those rows (inner join). That may hide legitimate data. | Scan for `<SELECT COUNT() FROM orders WHERE customer\_id IS NULL>`, `<SELECT COUNT() FROM order\_items WHERE order\_id IS NULL>`. |

| 3 | Duplicate keys | If `order\_id` or `customer\_id` are not unique, the result set will contain duplicates of the joined rows. |
`<SELECT order\_id, COUNT() FROM orders GROUP BY order\_id HAVING COUNT() > 1>`, same for customers. |

| 4 | Column correctness | Only the columns you list are fetched; ensure the dashboard really needs every one of them. | Interview the dashboard author or review the report spec. |

| 5 | Result?set equivalence | The new query must produce the same rows (or a superset if you add aggregates) as the original, but without the cross?join issue. | Run both queries in a dev sandbox against a representative dataset and compare row counts and checksums (`HASH\_AGG`). |

| 6 | Filter logic | The date and region predicates must still be applied correctly. | Verify that `order\_date >= '2024?01?01'` is using the correct data type (DATE) and that `region = 'EMEA'` is case?sensitive/collation?aware as expected. |

| 7 | Execution plan | The planner should now use the two predicates and join keys, resulting in a much smaller join tree. | Run `<EXPLAIN USING STATEMENT>` for the new query and confirm that: Orders and customers are filtered first.The join on `order\_id` is a hash?join or a nested?loop (depending on size).No large intermediate scans show up. |

| 8 | Cluster keys / Materialized Views | If your warehouse uses clustering on `order\_id` or `region`, the new query can exploit them. | Confirm that clustering exists with `<SHOW TABLES LIKE 'ORDERS'>` (look for `CLUSTER BY`). |

| 9 | Resource usage | Even with the join predicates, the join of 12?M orders × ~400?M items could still be heavy. | Execute the query in a sandbox warehouse of the same size and monitor `<SHOW WAREHOUSE> SHOWING` for credits and time. |

| 10 | Edge?case handling | Empty results, very large items per

Engineering counterpart

7:27 AM - Responded in 0s

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

Agent response

7:27 AM - Responded in 11s

Top Risks

| # | Risk | Impact | Likelihood |

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

| 1 | Missing/incorrect join predicates - If the foreign?key names are wrong or the relationships differ (e.g., `cust\_id` ? `customer\_id`), the query will silently drop rows or produce wrong results. | Incorrect sales data in the dashboard. | Medium |

| 2 | Null or duplicate keys - `NULL` or non?unique `order\_id`/`customer\_id` will cause missing or duplicated rows. | Distorted totals / reporting anomalies. | Medium |

| 3 | Performance regressions - If the dataset is larger than expected or clustering is absent, the join still may require large intermediate scans, potentially causing timeouts or credit spikes. | Dashboard timeout / higher costs. | Low?Medium |

| 4 | Column mismatch - The selective `SELECT` list may omit a column that the dashboard expects, causing errors or incomplete visuals. | Dashboard failure or incomplete data. | Low |

| 5 | Date / region filter mis?applied - Wrong data type or collation can change the filter results. | Wrong subset of orders, potentially missing data. | Low |

Rollback Plan

1. Keep the original statement in a versioned script

Example: `reports/slow\_sales\_draft\_v1.sql` (the current query).

Store the rewritten statement in a separate file `reports/slow\_sales\_draft\_v2.sql`.

2. Use a non?production (dev) warehouse for testing

Run both queries in the same dev warehouse.

Compare row counts, checksums, and execution plans.

If the rewritten query fails (timeout, error, or different results), stop the deployment and keep the original.

3. Automated regression test

Create a small test harness that runs both queries against a controlled snapshot of the tables.

If the rewritten query returns a different number of rows or a different aggregated total, flag an alert.

4. Feature?flag in the dashboard

Deploy the new query behind a feature flag or a parameter (`USE\_NEW\_QUERY = TRUE/FALSE`).

Initially set it to `FALSE`; enable for a subset of users or a short period.

If any user reports wrong data or timeouts, immediately flip the flag back.

5. Credit monitoring

Track credit usage before and after the switch.

If usage spikes beyond a defined threshold, abort and revert.

6. Rollback action

If rollback is needed, simply point the dashboard's SQL definition back to the original version (`SELECT ... FROM orders, ...`).

No data changes were performed, so no schema changes or deletes were involved-just a query switch.

Bottom line:

- Verify join key names and uniqueness before deployment.

- Run side?by?side tests in a dev warehouse.

- Keep the original query as a quick rollback and use a feature flag to roll out gradually.

- Monitor both correctness and performance; if anything goes wrong, simply toggle the flag back to the original query.

Engineering counterpart**7:27 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.