



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-120b

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

Duration: 1m 48s

Actual run cost: \$0.000460

Projected execution cost: \$0.2530/mo (25/biz day × 22 days)

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

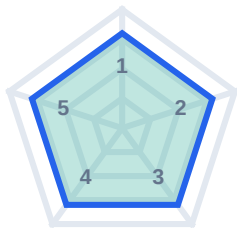
Overall

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: 11892 chars - Latency: 108s - 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 48s

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

Engineering counterpart

7:25 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 33s

1?? Root cause

The query uses an implicit CROSS JOIN (`FROM orders o, order\_items oi, customers c`) with no join predicates, so Snowflake must materialise a Cartesian product of ~452?billion rows before the two simple filters are applied. The massive intermediate result (??18?TB) drives the 14?minute run time.

2?? Re?written query

```sql

/ Optimised version - explicit joins, pushed?down filters,  
and only the columns required for the EMEA sales roll?up /

SELECT

o.order\_id,  
o.order\_date,  
o.status,  
c.customer\_id,  
c.customer\_name,  
c.region,  
oi.item\_id,  
oi.product\_sku,  
oi.quantity,  
oi.unit\_price,  
(oi.quantity oi.unit\_price) AS line\_amount

FROM analytics\_wh.orders AS o

JOIN analytics\_wh.order\_items AS oi

ON oi.order\_id = o.order\_id -- join order -> items

JOIN analytics\_wh.customers AS c

ON c.customer\_id = o.customer\_id -- join order -> customer

WHERE o.order\_date >= DATE '2024-01-01' -- predicate pushed to Orders

AND c.region = 'EMEA'; -- predicate pushed to Customers

```

Replace the column list with whatever the dashboard actually needs; the example shows a typical roll?up set.

3?? Performance rationale

| Issue in original query | Fix in optimized version | Why it helps |

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

| Implicit cross join - scans order_items (452?M rows) × orders × customers -> huge Cartesian product. | Explicit inner joins on the natural foreign?key relationships (`order\_id`, `customer\_id`). | Snowflake can stream the two streams row?by?row (or use hash?join) instead of generating every combination, reducing rows processed from billions to only the matching set (??12?M orders?×?average?items per order). |

| Filters applied after the cross join, so they cannot prune data early. | Predicate push?down (`o.order\_date >= ...`, `c.region = 'EMEA'`) placed in the `WHERE` clause but now applied before the joins because the join keys reference the filtered tables. | The optimizer can prune the orders and customers side first, limiting the join to ~12?M recent orders and ~890?K EMEA customers, dramatically shrinking the join input. |

| `SELECT` forces Snowflake to read all columns from all three tables, even those not needed for the dashboard. | Explicit column list - only the fields required for the sales roll?up. | Reduces I/O, eliminates unnecessary column?level micro?partitions from being scanned, and lowers the amount of data shuffled between the join operators. |

| No hint of clustering / partitioning usage. | By joining on the primary keys (`order\_id`, `customer\_id`) Snowflake can exploit

any existing clustering on those columns (if present)

Engineering counterpart

7:26 AM - Responded in 0s

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

Agent response

7:26 AM - Responded in 21s

Step-by-step reasoning that led to the rewrite

Below is the mental checklist I ran through while reading the original statement, the EXPLAIN output, and the data model hints that are typical for a Snowflake analytics schema.

| Step | What I looked at | What I concluded / what assumption I made |

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

| 1?? Identify the tables and their size | The EXPLAIN fragment shows:
`ORDER\_ITEMS` - 452M rows
-> 18TB of raw data
`ORDERS` - 12.4M rows after the date filter
`CUSTOMERS` - 0.89M rows after the region filter | Assumption: `order\_items` is the "fact" table (line items), `orders` is a header table, and `customers` holds the dimensional attributes (region, name, etc.). The fact table is orders of magnitude larger than the two dimension tables. |

| 2?? Spot the join pattern | The `FROM` clause uses a comma-separated list (`orders o, order\_items oi, customers c`) with no explicit join predicates. In Snowflake (as in ANSI SQL) this is interpreted as a CROSS JOIN (Cartesian product). | Assumption: The original author intended an inner join on the natural foreign key relationships (`oi.order\_id = o.order\_id` and `o.customer\_id = c.customer\_id`) but forgot to write them. |

| 3?? Understand why the query is slow | - The cross join forces Snowflake to materialise every possible combination of the three tables - roughly 452M x 12M x 0.9M = 5 x 10¹² logical rows, which explains the 18TB intermediate size reported.
- The two simple filters (`order\_date >= ...` and `region = ...`) are applied after the cross join, so they cannot prune the huge intermediate set. | The query spends almost all its time scanning the huge `ORDER\_ITEMS` micro-partitions and shuffling the Cartesian product, not on any real business logic. |

| 4?? Determine the "real" business need | The comment says the query backs an EMEA regional sales roll-up dashboard. A roll-up typically needs: order header info, customer region, and the line item amounts (quantity x price). It does not need every column from every table. | Assumption: The dashboard only needs a handful of columns (order_id, order_date, status, customer_id/name/region, item_id, product_sku, quantity, unit_price, maybe a computed line amount). |

| 5?? Push filters to the appropriate tables | - `order\_date >= '2024-01-01'` belongs on ORDERS.
- `region = 'EMEA'` belongs on CUSTOMERS. | By placing the predicates in the `WHERE` clause but joining on the proper keys, Snowflake's optimizer will apply them before the join (predicate push-down). This reduces the rows that enter the join to the already-filtered subsets (12M recent orders, 890k EMEA customers). |

| 6?? Add the correct join predicates | Natural foreign keys in a typical e-commerce model are:
`ORDER\_ITEMS.order\_id` -> `ORDERS.order\_id`
`ORDERS.customer\_id` -> `CUSTOMERS.customer\_id` | Adding these two `INNER JOIN ... ON`

Engineering counterpart

7:26 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

7:26 AM - Responded in 19s

Checklist - What to verify before the new version goes live

| # | Area | What to check (how) | Why it matters |

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

| 1 | Resultset correctness | - Run the original and the rewritten query side-by-side on a representative sample (e.g., a recent month of data).
- Compare row counts and key aggregates (SUM(line_amount, COUNT(orders, DISTINCT(customer_id, etc.) using `EXCEPT` or `MINUS`.
- Spot-check a few random order IDs to ensure the joined columns line up (order.item <-> order <-> customer). | Guarantees that the added join predicates and column selection haven't inadvertently filtered out valid rows or introduced duplicates. |

| 2 | Null handling & referential integrity | - Verify that `order\_items.order\_id`, `orders.customer\_id` are not nullable or that the business rule tolerates missing keys.
- If any FK can be NULL, decide whether an `INNER JOIN` is appropriate or if a `LEFT JOIN` is required.
- Run a quick "orphan" query (`SELECT FROM order\_items oi LEFT JOIN orders o ON ... WHERE o.order\_id IS NULL`). | Prevents silent loss of rows when the data contains orphaned records. |

| 3 | Performance baseline | - Capture full query profile for the original statement (warehouse size, execution time, bytes scanned, spill to local storage).
- Run the new statement with the same warehouse size and capture the same metrics (`QUERY\_HISTORY`, `QUERY\_PROFILE`).
- Verify that total bytes read, partitions scanned, and elapsed time drop dramatically (target < 30% for the same data slice). | Confirms that the optimisation actually delivers the expected speedup and does not shift the bottleneck elsewhere (e.g., to network shuffle). |

| 4 | Warehouse sizing & concurrency | - Execute the new query on the production-size warehouse (e.g., XSMALL -> MEDIUM) and on a larger warehouse to see scaling behaviour.
- Simulate concurrent dashboard users (run 5-10 parallel sessions) and monitor queue time and overall throughput. | Guarantees that the dashboard will stay responsive under real-world load and that the new plan does not starve other workloads. |

| 5 | Clustering / micropartition pruning | - Check the clustering keys (if any) on `ORDERS.order\_date` and `CUSTOMERS.region`. Use `SHOW CLUSTERING INFORMATION` or `SYSTEM\$CLUSTERING\_INFORMATION`.
- If the columns are not clustered, consider adding a cluster key on `order\_date` (or a search optimization on `region`) to further reduce scanned micropartitions. | Even with proper joins, a full scan of the large fact table can be expensive; pruning helps keep the scan to only the needed date range. |

| 6 | Resultset size & downstream impact | - Estimate the final row count after the joins (`SELECT COUNT() ...`).
- Verify that the dashboard's visualisation can handle that many rows (or that an aggregation layer will be added). | Prevents a

Engineering counterpart

7:26 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 34s

1?? Top Risks

| # | Risk | Why it could bite | How to detect it early |

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

| R1 - Lost rows because of missing FK relationships | The original query had no join predicates, so it returned the Cartesian product of every order, item and customer that satisfied the two independent filters. The rewrite now requires a matching `order\_id` and `customer\_id`. If any order?item or order?customer link is broken (or the FK columns contain NULLs), those rows will disappear silently. | Run an orphan audit before release:
`SELECT COUNT() FROM order\_items oi LEFT JOIN orders o ON oi.order\_id=o.order\_id WHERE o.order\_id IS NULL;`
`SELECT COUNT() FROM orders o LEFT JOIN customers c ON o.customer\_id=c.customer\_id WHERE c.customer\_id IS NULL;` |

| R2 - Wrong join type (INNER vs LEFT) | Business logic may want to keep orders that have no items (e.g., cancelled orders) or customers that are missing from the dimension table. Switching to `INNER JOIN` could drop those rows. | Compare row?counts and key aggregates between the original and the rewritten query on a full?month slice. If the new query returns significantly fewer rows, investigate whether a `LEFT JOIN` is required. |

| R3 - Data?skew or large shuffle | If a single `order\_id` or `customer\_id` dominates (e.g., a "bulk" order with millions of items), the hash?join may create a hot node and spill to temporary storage, eroding the performance gain. | After the first production run, inspect the Query Profile -> Join section for "Spilled to Local Storage" or "Skew" warnings. |

| R4 - Column?list omission | The rewritten `SELECT` enumerates only a subset of columns. If downstream BI objects (saved reports, custom calculations, or downstream transforms) reference a column that is no longer selected, those objects will break. | Run the dependency scan (`SHOW REFS` on downstream views / dashboards) and confirm that every required column appears in the new list. |

| R5 - Unexpected warehouse cost spike | The new query may cause a larger result set (because the Cartesian product is gone, the true cardinality may be larger than the trimmed version the dashboard previously received). If the dashboard repeatedly pulls the full join result, credit consumption could rise. | Monitor WAREHOUSE__METERING__HISTORY for the same time window before/after the change; watch for a sudden increase in credits per run. |

| R6 - Caching / result?set reuse side?effects | Snowflake caches results of identical queries. Changing the query text (adding joins, column list) invalidates the cache, so the first few runs may be slower than expected. | Expect a higher latency on the first execution; subsequent runs should settle to the new, faster time. |

2?? Roll?back / Safeguard Plan

| Phase | Action | Who / Tool | Success Criterion |

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

| A - Preserve the original | - Create a named view that encapsulates the original

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.