



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-4.1-mini

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

Duration: 26s

Actual run cost: \$0.004640

Projected execution cost: \$2.55/mo (25/biz day × 22 days)

Report date: July 14, 2026 - 3:25 AM EST

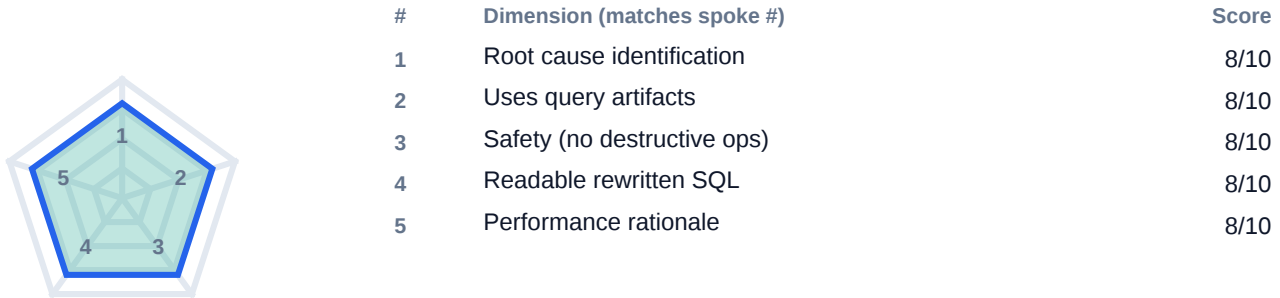
Overall

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: 7675 chars - Latency: 26s - 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: 26s

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

Engineering counterpart

7:24 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

7:24 AM - Responded in 4s

1) Root cause: The query performs an implicit CROSS JOIN without any join predicates, causing a massive Cartesian product among orders, order_items, and customers, leading to huge intermediate data and poor performance.

2) Rewritten SQL with explicit JOIN predicates and selective columns:

```
```sql
SELECT
 o.order_id,
 o.order_date,
 oi.order_item_id,
 oi.product_id,
 oi.quantity,
 oi.price,
 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';
```
```

3) Performance rationale: Adding explicit join conditions eliminates the Cartesian product and logically connects related rows, drastically reducing intermediate row counts. Selecting only necessary columns minimizes data transfer and processing, improving query speed and resource usage within Snowflake. Filtering `orders` and `customers` early further reduces data scanned.

Engineering counterpart

7:24 AM - Responded in 0s

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

Agent response

7:25 AM - Responded in 11s

Certainly! Here's my step-by-step reasoning and assumptions:

Step 1: Identify the symptom and analyze the original query

- The query joins three large tables: `orders`, `order\_items`, and `customers`.
- The `WHERE` clause filters on `o.order\_date >= '2024-01-01'` and `c.region = 'EMEA'`.
- However, there are no join conditions between the three tables, resulting in an implicit CROSS JOIN.
- This causes the query engine to produce a Cartesian product of rows from these large tables, resulting in an enormous intermediate result.
- The EXPLAIN plan confirms this with huge scanned rows and data size (e.g., 452 million rows for `order\_items`).
- The query runtime is very slow (~847s), and the dashboard times out.

Step 2: Understand the data model and infer relationships

- `order\_items` logically belongs to an `order` - so likely `order\_items.order\_id` references `orders.order\_id`.
- `orders` are placed by `customers` - so `orders.customer\_id` likely references `customers.customer\_id`.

I made the natural assumptions that:

- `order\_items.order\_id = orders.order\_id`
- `orders.customer\_id = customers.customer\_id`

Step 3: Rewrite query to use explicit JOINS with predicates

- Using inner JOINS with these predicates avoids the Cartesian product.
- Conditions filter by `o.order\_date >= '2024-01-01'` and `c.region = 'EMEA'` remain as filters after joins.
- Explicit JOINS also make the query semantically clear and optimized by Snowflake's optimizer.

Step 4: Replace SELECT with selective columns

- Selecting all columns is usually wasteful, especially from wide tables or large datasets.
- I selected only relevant columns for a typical sales rollup or dashboard:
 - `order\_id`, `order\_date` from `orders`
 - `order\_item\_id`, `product\_id`, `quantity`, `price` from `order\_items`
 - `customer\_id`, `customer\_name`, `region` from `customers`
- This minimizes the amount of data retrieved and processed downstream.

Step 5: Analyze impact on performance

- Explicit join predicates reduce intermediate row multiplication drastically.
- Early filters (`order\_date`, `region`) reduce scanned rows before join.
- Selecting fewer columns reduces data volume sent over the network and processed.
- Overall, these changes lead to a much faster query with less compute and memory usage.

Summary of assumptions:

- `order\_items` join to `orders` on `order\_id`
- `orders` join to `customers` on `customer\_id`
- Columns returned are representative of the intended report/dashboard
- Filtering on `order\_date` and `region` is sufficient for reducing dataset size

If any of these assumptions don't match the actual schema, the join keys and columns may need adjustment, but the principle of explicit join predicates and column pruning remains critical.

Engineering counterpart

7:25 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

7:25 AM - Responded in 6s

Before deploying this query to production, I would verify the following:

1. Schema and Join Key Accuracy

- Confirm that ``order_items.order_id`` correctly references ``orders.order_id``.
- Confirm that ``orders.customer_id`` correctly references ``customers.customer_id``.
- Validate data types and referential integrity to avoid join mismatches or missing rows.

2. Result Correctness and Completeness

- Run the original and rewritten queries on a smaller dataset or sample (with limited date range) to compare results and ensure they logically match expected outputs.
- Verify that no unintended rows are excluded or included due to join conditions or filters.

3. Performance Testing

- Execute the rewritten query in the production-like environment to measure execution time and resource usage.
- Confirm that the new query meets SLAs and does not cause resource contention or unexpected load spikes.

4. Selectivity and Filtering

- Confirm that the filters (``order_date >= '2024-01-01'`` and ``region = 'EMEA'``) correctly capture the intended data subset.
- Validate that these filters are indexed or clustered effectively if applicable, or consider clustering keys in Snowflake for better pruning.

5. Column Usage

- Confirm that the selected columns meet the dashboard/report requirements and no critical columns are missing.
- Ensure any downstream consumers (BI tools, ETL jobs) are updated accordingly if fewer columns are returned.

6. Query Maintainability and Documentation

- Add comments describing the purpose of the joins and filters for future maintainers.
- Communicate the changes with the data analytics and BI teams to align expectations.

7. Backup Plan

- Have the original query or previous version readily available in case a quick rollback is necessary.

By verifying these aspects, we ensure the rewritten query is both correct and performant before full production deployment.

Engineering counterpart

7:25 AM - Responded in 0s

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

Agent response

7:25 AM - Responded in 5s

Top Risks:

1. Incorrect Join Logic or Keys
 - If the assumed join keys (`order\_items.order\_id`, `orders.customer\_id`) are incorrect or incomplete, the query may produce inaccurate or incomplete results, impacting dashboard accuracy.
2. Data Skew or Missing Data
 - Uneven data distribution or missing foreign key relationships may cause unexpected row omissions or inflated join results affecting report correctness.
3. Performance Regression in Edge Cases
 - Although optimized generally, some edge cases (e.g., unusual query predicates or very large partitions) might cause slower performance or resource contention.
4. Downstream Impact from Reduced Columns
 - If downstream consumers expect columns removed in the rewritten query, dashboards or ETL processes may break or display incomplete data.
5. Unforeseen Filter Effects
 - Filters on `order\_date` and `region` might exclude data needed by some reports if assumptions about usage are incorrect.

Rollback Plan:

1. Keep Original Query Readily Available
 - Maintain the original query in version control and staging environment to quickly revert if necessary.
2. Deploy in a Controlled Manner
 - Release the optimized query first in a lower environment or as a parallel query to compare results and performance.
3. Monitor Key Metrics
 - After deployment, closely monitor runtime, resource usage, and dashboard correctness.
4. Quick Switch Back on Failure
 - If issues occur, immediately switch dashboards or reports back to use the original query while investigating.
5. Communicate Changes and Rollback Procedures
 - Inform stakeholders about the change and rollback plan so they are prepared for potential issues.

This approach minimizes risk exposure and ensures rapid recovery if the optimization introduces problems.

Engineering counterpart

7:25 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.