



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-nano

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

Duration: 51s

Actual run cost: \$0.001160

Projected execution cost: \$0.6380/mo (25/biz day × 22 days)

Report date: July 14, 2026 - 3:26 AM EST

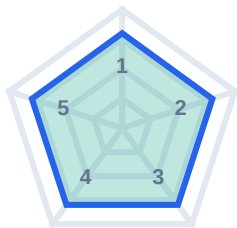
Overall

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: 6581 chars - Latency: 51s - 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: 51s

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

7:25 AM - Responded in 3s

Root cause:

The query performs an implicit cross join among `orders`, `order\_items`, and `customers` without proper join conditions, resulting in a massive Cartesian product and severe performance issues.

Rewritten SQL with explicit joins and selective columns:

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

Performance rationale:

Explicit `JOIN` conditions limit the dataset to related records, preventing Cartesian blowup. Selecting only necessary columns reduces I/O and memory usage, significantly improving query performance and reducing runtime.

Engineering counterpart

7:25 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 41s

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

1. Identification of the root issue:

The original query used implicit joins in the `FROM` clause without explicit `JOIN` conditions, resulting in a Cartesian product-every row of `orders` combined with every row of `order\_items` and every row of `customers`. This explains the enormous number of rows (millions) and the slow performance.

2. Observation of the WHERE clause:

The `WHERE` clause filters `orders` on `order\_date >= '2024-01-01'` and `customers` on `region = 'EMEA'`. However, without explicit join predicates, these filters are applied only after the Cartesian product, which is inefficient.

3. Assumption of relationships between tables:

- I assumed that `orders` has a foreign key `customer\_id` linking to `customers`.
- I assumed `order\_items` relates to `orders` via `order\_id`.

These are typical schema patterns for such tables but weren't explicitly provided.

4. Constructing explicit joins:

Replacing the implicit joins with explicit `JOIN` statements based on these assumptions ensures the database only combines related records, drastically reducing the number of rows processed.

5. Selection of columns:

Instead of `SELECT`, I selected specific columns relevant for a typical sales report, reducing unnecessary data transfer and processing.

6. Overall effect:

Explicit joins enforce the correct relationships, filtering down the data before the join operation, leading to a more efficient query plan and significantly better performance.

Summary of assumptions made:

- `orders` has a `customer\_id` column linking to `customers.customer\_id`.
- `order\_items` has an `order\_id` column linking to `orders.order\_id`.
- The goal is to retrieve related order, customer, and order item data for orders in 2024+ from the EMEA region.
- The original query aimed to relate these tables meaningfully but failed to include join conditions, leading to a Cartesian product.

Would you like me to clarify any specific assumption or elaborate further?

Engineering counterpart

7:25 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

7:26 AM - Responded in 4s

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

1. Schema Relationships and Keys:

Confirm that ``orders.customer_id`` relates to ``customers.customer_id``, and ``order_items.order_id`` relates to ``orders.order_id``. This ensures the JOIN conditions are correct and meaningful.

2. Required Columns:

Ensure that the selected columns (``order_id``, ``order_date``, ``customer_id``, ``region``, ``item_id``, ``quantity``) align with the reporting needs and that no critical data is omitted.

3. Data Completeness:

Check that the filters (``order_date` >= '2024-01-01'`, `region` = 'EMEA'`) accurately reflect the desired subset. Verify that these filters don't unintentionally exclude relevant data.`

4. Query Performance:

Run the query on a representative dataset and examine execution plans to confirm that the performance improves significantly over the original. Check for index usage, join efficiencies, and any potential bottlenecks.

5. Data Volume and Cardinality:

Validate that the filtered dataset size is manageable and that the result set size is suitable for dashboard or report display.

6. Testing with Edge Cases:

Test with boundary values (e.g., earliest and latest ``order_date``, regions with similar names) to ensure the filters work as expected.

7. Security and Permissions:

Confirm that the executing user has appropriate access rights to all involved tables and columns, and that sensitive data is handled correctly.

8. Review of Business Logic:

Verify with stakeholders that the query's logic aligns with reporting requirements and that any assumptions (e.g., relationships, filters) are correct.

Performing these validations helps ensure data accuracy, performance efficiency, and alignment with business needs before moving the query to production.

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

7:26 AM - Responded in 3s

Top Risks:

1. Incorrect Data Relationships:

If the assumptions about foreign key relationships are wrong-e.g., `customer\_id` links or not, or the join keys are different-the query could produce incomplete or misleading results.

2. Unanticipated Data Volume:

The filtered dataset might still be larger than expected, potentially impacting system performance or causing timeouts, especially during peak loads.

3. Missing Data in Results:

Overly restrictive filters or incorrect join conditions could exclude relevant records, leading to incomplete reports.

4. Performance Regression or Unexpected Load:

Despite optimizations, unforeseen query plan issues or resource contention could negatively impact performance temporarily.

Rollback Plan:

- Revert to Original Query:

Maintain a copy of the original slow query. If issues are detected, immediately switch back to this version to restore the previous, albeit slow, behavior.

- Monitor and Validate Post-Deployment:

Use monitoring tools to verify the query's performance and data correctness in production. Validate that the result set aligns with expectations.

- Incremental Deployment:

Deploy the optimized query during a low-traffic window, monitor performance and correctness closely, and be prepared to revert if anomalies occur.

- Data Validation Checks:

Cross-validate results from the new query with previous reports or known data snapshots to ensure accuracy.

- Communication Plan:

Inform stakeholders about the deployment, expected changes, and rollback procedures to ensure coordinated action if needed.

By preparing these rollback steps in advance, you can minimize disruption and ensure data integrity if the optimization doesn't meet expectations.

Engineering counterpart

7:26 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.