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-4o-mini
Roles: Data Engineer (AI agent) <> Engineering counterpart (AI agent)
Duration: 37s
Actual run cost: $0.001740
Projected execution cost: $0.9570/mo (25/biz day × 22 days)
Report date: July 13, 2026 - 11:21 PM EST

Session: sample-lb-rankings-seed-df185bf8-3380-4e50-a529-072d44a43326

de_sql_optimization_v1 - agent - openrouter - openai/gpt-4o-mini - ended

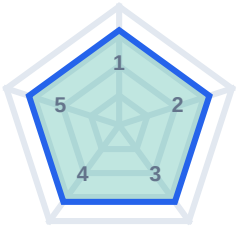
Overall

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: 9259 chars - Latency: 37s - 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: 37s

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

Engineering counterpart
3:20 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
3:20 AM - Responded in 4s

1) Root cause: The query is performing an implicit cross join between the `orders`, `order_items`, and `customers` tables due to the lack of explicit join conditions, resulting in a Cartesian product and a massive increase in the number of rows processed.

2) Rewritten SQL:
```sql
SELECT o.order_id, o.order_date, oi.item_id, oi.quantity, c.customer_id, 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) Brief performance rationale: By explicitly joining the tables using their primary key and foreign key relationships, the query significantly reduces the number of rows processed. This helps eliminate the Cartesian product, leading to improved performance by focusing only on the relevant records that meet the specified criteria. Selecting only necessary columns further reduces the amount of data transferred and processed.

Engineering counterpart
3:20 AM - Responded in 0s
Thanks - walk me through your reasoning step by step. What assumptions did you make?

Agent response
3:20 AM - Responded in 8s
Certainly! Here's a step-by-step breakdown of the reasoning and assumptions made during the optimization:

1. Identifying the Problem:
- The original query utilized an implicit join (via comma-separated tables in the FROM clause), which results in a Cartesian product when there are no explicit JOIN conditions. This increases the total number of rows processed dramatically, contributing to the long execution time.

2. Understanding the Structure of the Data:
- I assumed that there are common keys between the `orders`, `order_items`, and `customers` tables:
  - `orders` likely has a foreign key to `customers`, typically `customer_id`.
  - `order_items` has a foreign key to `orders`, typically `order_id`.
- These assumptions are based on standard database design practices where orders are linked to customers and orders have associated items.

3. Clarifying the Query Requirements:
- The query needs to filter orders by date and customers by region - specifically, orders made after January 1, 2024, and customers located in the 'EMEA' region.
- This was identified as the primary selection criteria to keep the query focused and efficient.

4. Formulating the Explicit Joins:
- I restructured the query to use `JOIN` syntax, making it clear how the tables relate to one another. This is important for:
  - Clarifying the relationships between tables.
  - Improving performance by allowing the query optimizer to more effectively execute the query plan.
- Explicit joins also reduce potential confusion and errors in understanding how data is combined.

5. Selecting Specific Columns:
- To optimize the query further, I replaced `SELECT ` with specific columns. This practice has several benefits:
  - It reduces the amount of data processed and transferred, thereby decreasing runtime.
  - It enhances readability, making it clear which fields are necessary for the analysis or report.

6. Rationale for Performance Improvement:
- By refining the joins and filtering the results upfront, the database engine can apply indexing and other optimization techniques more efficiently. As such, the execution plan generated for the rewritten query would be expected to handle a significantly smaller dataset.

Overall, the optimization focused on leveraging the inherent relationships within the data for efficiency, alongside best practices in writing SQL queries. The assumptions made were based on typical database design patterns and common practices observed in SQL optimization.

Engineering counterpart
3:20 AM - Responded in 0s
What would you verify before shipping this to production?

Agent response
3:20 AM - Responded in 9s

Before shipping the optimized SQL query to production, several key aspects should be verified to ensure that the query performs well and produces the correct results. Here's a checklist of what to confirm:

1. Test the Query Performance:
   - Run the optimized query in a development or staging environment to compare its execution time against the original query. Ensure that the performance improvement is significant enough to justify the change.

2. Validate the Output:
   - Compare the results of the optimized query against the original query. Check that the output is consistent (i.e., same number of rows and the same data for each row) to ensure no data has been inadvertently filtered out or omitted.

3. Check Execution Plans:
   - Review the execution plan for the optimized query using the `EXPLAIN` statement. Confirm that the joins are being handled as intended and that there are no significant unexpected operations (e.g., large scans or unnecessary joins).

4. Indexing and Statistics:
   - Ensure that the relevant columns involved in the joins and filters (such as `order_id`, `customer_id`, `order_date`, and `region`) are indexed appropriately. Check if statistics are up-to-date to aid the query optimizer in generating an efficient execution plan.

5. Test with Different Data Volumes:
   - Validate the query against various data sets, including edge cases (e.g., very small or large datasets) to ensure that performance remains acceptable across different scenarios.

6. Error Handling:
   - Review the query for potential error handling or edge cases, such as missing data, null values, or unexpected data types that could cause issues during execution.

7. Impact on Resources:
   - Assess the resource usage (CPU, memory, and I/O) of the query in the testing environment. Make sure it does not exceed acceptable limits for the production environment.

8. Review by Peers:
   - Have the query reviewed by peers or team members for additional validation and to catch any potential oversights.

9. Monitoring Plan:
   - Establish monitoring for the query once it goes into production to track its performance over time. Set up alerts for any significant changes in execution time, row counts, or system resource usage.

10. Documentation:
    - Update any relevant documentation concerning the query's purpose, its structure, and performance improvements. This helps ensure that future developers understand the changes made.

By confirming these aspects, you can reduce the risk of introducing issues into the production environment and ensure the new query performs optimally.

Engineering counterpart
3:20 AM - Responded in 0s

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

Agent response
3:21 AM - Responded in 16s

When deploying an optimized SQL query to production, it's important to be aware of potential risks and have a rollback plan in place. Here are the top risks and a corresponding rollback strategy:

Top Risks

1. Incorrect Results:
   - The optimized query might yield different results compared to the original, possibly due to changes in filtering, joins, or data assumptions.

2. Performance Regression:
   - Although the query is expected to perform better, it could still perform worse under certain conditions, especially with varying data volumes.

3. Resource Overutilization:
   - The new query might consume more system resources (CPU, memory, and I/O) than anticipated, leading to contention with other processes or affecting overall system performance.

4. Longer Execution Time in Edge Cases:
   - For some data distributions or edge cases, the new query execution time could be longer than that of the original query.

5. Impact on Dependent Processes:
   - If other processes, reports, or dashboards depend on the output of this query, changes in the results could affect their accuracy and function.

6. Lack of Backward Compatibility:
   - The optimized query may not be compatible with existing tools or applications that expect the original schema or output.

Rollback Plan

1. Version Control:
   - Maintain a version history of the original query and the optimized query in a version control system (e.g., Git). This allows easy retrieval of the original version if needed.

2. Testing Environment:
   - Before deployment, test the original and optimized queries side by side in a staging environment, comparing results and performance metrics. If discrepancies arise post-deployment, revert to the original.

3. Immediate Rollback Process:
   - In case of detected issues after deployment, have a procedure to quickly switch back to the original query. This may involve commenting out the optimized query in the production environment and uncommenting the original.

4. Monitoring:
   - Set up monitoring to track key performance indicators (KPIs) associated with the query's execution. If performance degrades or errors occur, alert the team and revert to the original query.

5. Grace Period:
   - Implement a grace period after deployment during which both queries can run simultaneously (if feasible), allowing for live comparisons without impacting users.

6. Communication Plan:
   - Communicate with stakeholders about potential issues and the availability of the rollback plan. Ensure that the team knows how to execute the rollback if necessary.

7. Post-Rollback Analysis:
   - If a rollback occurs, conduct a thorough analysis of what went wrong, including reviewing execution plans, logs, and any relevant metrics. This analysis should inform further iterations or revisions of the optimization.

By identifying these risks and having a clear rollback plan, you can minimize disruptions and maintain confidence in the production environment when deploying the optimized SQL query.

Engineering counterpart
3:21 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.