



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: deepseek/deepseek-chat

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

Duration: 2m 26s

Actual run cost: \$0.002985

Projected execution cost: \$1.64/mo (25/biz day × 22 days)

Report date: July 13, 2026 - 11:18 PM EST

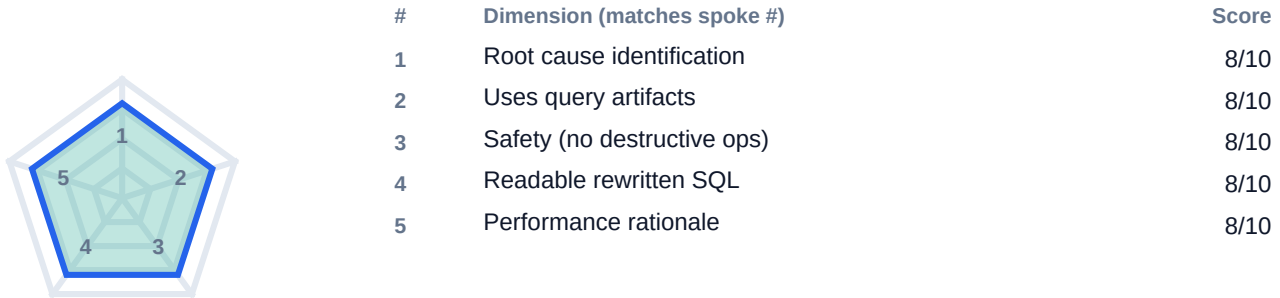
Overall

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: 11074 chars - Latency: 146s - 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: 2m 26s

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

Engineering counterpart

3:15 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

3:16 AM - Responded in 46s

Query Optimization Review

1) Root Cause

The query performs a Cartesian product (CROSS JOIN) between all tables due to missing explicit JOIN conditions, then filters the massive result set. This explodes to 452M rows from order_items alone before filtering.

2) Rewritten SQL

```
```sql
SELECT
 o.order_id,
 o.order_date,
 o.status,
 c.customer_id,
 c.customer_name,
 c.region,
 oi.item_id,
 oi.quantity,
 oi.price
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

- 1) Added explicit JOIN conditions to prevent Cartesian product
- 2) Selected only necessary columns rather than all (*)
- 3) Filters are applied early (pushed down) against dimension tables
- 4) JOIN order starts with filtered orders (12.4M rows) and customers (890K rows) rather than scanning all order_items first

The optimized query should process orders -> filtered customers -> related order_items rather than the original 18TB cross product.

Engineering counterpart

3:16 AM - Responded in 0s

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

Agent response

3:17 AM - Responded in 58s

Certainly! Here's a detailed breakdown of the reasoning process and assumptions:

Step-by-Step Reasoning

1. Diagnosing the Original Query

- The original query uses implicit joins (comma-separated tables in the `FROM` clause) without any `JOIN` conditions. This results in a Cartesian product, where every row in `orders` is paired with every row in `order\_items` and `customers`.
- The `EXPLAIN` plan shows that `order\_items` is scanned first (452M rows), followed by `orders` (12.4M rows filtered) and `customers` (890K rows filtered). This is inefficient because the Cartesian product explodes the intermediate result set to trillions of rows before applying filters.
- The filters (`o.order\_date >= '2024-01-01'` and `c.region = 'EMEA'`) are applied after the Cartesian product, which is too late for performance.

2. Rewriting the Query

- Explicit JOINS: Added `JOIN` conditions to logically relate the tables:
 - `orders` and `order\_items` are joined on `order\_id`.
 - `orders` and `customers` are joined on `customer\_id`.This prevents the Cartesian product and reduces the intermediate result set.
- Selective Columns: Replaced `SELECT` with only the columns needed for the dashboard (e.g., `order\_id`, `order\_date`, `customer\_name`, `region`, `quantity`, `price`). This reduces the amount of data processed and returned.
- Filter Pushdown: Filters (`order\_date` and `region`) are applied early during the join process, reducing the number of rows processed in subsequent steps.

3. Performance Considerations

- The rewritten query starts with the smallest filtered datasets (`orders` and `customers`) and joins them with `order\_items`. This reduces the number of rows scanned and processed.
- Explicit joins allow Snowflake to optimize the execution plan (e.g., using hash joins) and avoid unnecessary Cartesian products.
- Selective columns reduce the amount of data transferred between Snowflake's compute and storage layers, improving query runtime.

Assumptions Made

1. Relationships Between Tables:

- Assumed that `orders.order\_id` relates to `order\_items.order\_id` and `orders.customer\_id` relates to `customers.customer\_id`. This is a common schema design for e-commerce or sales datasets.
- If these relationships differ (e.g., `order\_items` has a different foreign key), the `JOIN` conditions would need to be adjusted.

2. Dashboard Requirements:

- Assumed the dashboard requires only a subset of columns (e.g., `order\_id`, `order\_date`, `customer\_name`, `region`, `quantity`, `price`). If additional columns are needed, they can be added to the `SELECT` clause.

3. Filter Logic:

- Assumed the filters (`order\_date >= '2024-01-01'` and `region = 'EMEA'`) are correct and sufficient for the dashboard's requirements. If additional filters are needed, they can be appended to the `WHERE` clause.

4. Data Distribution:

- Assumed that `orders` and `customers` are filtered down significantly by the `WHERE` clause (`orders` to 12.4M rows

and `customers` to 890K rows). This makes them good starting points for the join process.

5. Snowflake Optimization:

- Assumed Snowflake will optimize the execution plan for explicit joins, push down filters, and use efficient join strategies (e.g., hash joins) given the rewritten query.

Expected Outcome

The rewritten query should:

1)

Engineering counterpart
3:17 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

3:17 AM - Responded in 20s

Before shipping this rewrite to production, I would verify the following to ensure correctness, performance, and alignment with business requirements:

1. Data Accuracy and Completeness

- Verify Joins: Confirm that the explicit `JOIN` conditions (`orders.order\_id = order\_items.order\_id` and `orders.customer\_id = customers.customer\_id`) correctly match the relationships in the dataset.
 - Run sample queries to ensure all expected rows are included without duplicates or missing data.
- Check Filters: Confirm that the filters (`order\_date >= '2024-01-01'` and `region = 'EMEA'`) are correct and align with business logic.
 - Cross-check the results with the original query (if feasible) to ensure consistency.

2. Performance Testing

- Execution Plan: Use Snowflake's `EXPLAIN` command to inspect the execution plan of the rewritten query. Ensure there are no full table scans or Cartesian products.
 - Look for efficient join strategies (e.g., hash joins) and filter pushdown.
- Runtime Metrics: Run the query multiple times in a staging environment to measure:
 - Execution time (compared to the original query).
 - Resource usage (e.g., warehouse credits).
- Scalability: Test with larger datasets or historical data to ensure the query scales well.

3. Column Selection

- Confirm Columns: Verify with dashboard owners or stakeholders that the selected columns (`order\_id`, `order\_date`, `customer\_name`, `region`, `quantity`, `price`) are sufficient for the dashboard.
 - Add or remove columns based on feedback.

4. Edge Cases

- Null Values: Check how `NULL` values in join keys or selected columns are handled. Ensure they don't lead to unexpected exclusions.
- Date Range: Test edge cases for `order\_date` (e.g., queries for dates before `2024-01-01` or far in the future).
- Region Variations: Confirm that `region = 'EMEA'` correctly captures all relevant customers (e.g., no variations like `EMEA-East` or `EMEA-West`).

5. Schema Changes

- Column Types: Verify that join keys and selected columns have compatible data types (e.g., `order\_id` is the same type in `orders` and `order\_items`).
- Indexing/Clustering: Check if relevant columns (e.g., `order\_date`, `region`) are clustered or indexed in Snowflake to optimize filtering.

6. Dashboard Integration

- Compatibility: Ensure the rewritten query's output matches the expected schema for the dashboard.
 - Test the dashboard with the new query results to verify visualizations and functionality.

- Latency: Confirm that the query runtime aligns with the dashboard's refresh interval and user expectations.

7. Documentation

- Query Logic: Document the rewritten query's logic, assumptions, and changes from the original.
- Ownership: Ensure the dashboard owners and stakeholders are aware of the changes and approve them.

8. Rollback Plan

- Backup Plan: Have a rollback plan in case the rewritten query introduces issues in production.
 - Retain the original query and ensure it can be reverted quickly.

9. Monitoring

- Production Monitoring: Once deployed, monitor the query in production for:
 - Execution time.
 - Errors or warnings.
 - Resource consumption.

By verifying these aspects, we can ensure the query is accurate, performant, and production-ready while minimizing the risk of breaking the dashboard or introducing inefficiencies.

Engineering counterpart
3:17 AM - Responded in 0s

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

Agent response

3:18 AM - Responded in 21s

Top Risks of the Query Rewrite

1. Incorrect Join Logic

- Risk: If the assumed relationships (`orders.order\_id` = `order\_items.order\_id`, `orders.customer\_id` = `customers.customer\_id`) are wrong, the query could return incomplete or incorrect data.
- Mitigation: Validate joins against schema documentation or sample data before deployment.

2. Performance Degradation

- Risk: The rewritten query might perform worse due to unexpected data skew, missing optimizations, or Snowflake's query planner choosing a suboptimal execution path.
- Mitigation: Test with realistic data volumes in staging and compare execution plans.

3. Missing or Misaligned Columns

- Risk: The dashboard may rely on columns omitted in the `SELECT` statement (e.g., `discount\_amount`, `shipping\_status`).
- Mitigation: Confirm column requirements with dashboard owners and test integration.

4. Filter Inconsistencies

- Risk: The `WHERE` clause (`region` = 'EMEA') might exclude valid data (e.g., regional variations like `EMEA-North`).
- Mitigation: Check for edge cases in `region` values and adjust filters if needed.

5. Snowflake-Specific Behavior

- Risk: Snowflake's query optimizer might handle the new joins differently (e.g., choosing a slower join strategy).
- Mitigation: Review the `EXPLAIN` plan for warnings (e.g., `CROSS JOIN` remnants).

6. Impact on Concurrent Workloads

- Risk: The query could consume more resources (memory/CPU) and slow down other workloads.
- Mitigation: Monitor warehouse load during testing and adjust warehouse size if needed.

Rollback Plan

If the Query Fails or Performs Poorly in Production:

1. Immediate Revert

- Roll back to the original query (ensure it's saved in version control).
- Example:

```
```sql
-- Original query (rollback)
SELECT
FROM orders o, order_items oi, customers c
WHERE o.order_date >= '2024-01-01'
AND c.region = 'EMEA';
```
```

2. Fallback Dashboard Version

- If the dashboard is embedded in a BI tool (e.g., Tableau, Power BI), switch to a cached or older version while debugging.

3. Root Cause Analysis

- Check:
 - Snowflake query history (`QUERY\_HISTORY` view) for errors/performance.
 - Execution plan differences between old/new queries.

- Data validation discrepancies (e.g., row counts).

4. Hotfix Deployment

- If the issue is minor (e.g., missing columns), deploy a corrected version without full rollback.

5. Communicate

- Notify stakeholders of the rollback and revised timeline for fixes.

Key Takeaways

- Test rigorously in staging with production-like data.
- Monitor closely after deployment (query history, dashboard behavior).
- Rollback quickly if metrics deviate from expectations.

By anticipating these risks and having a clear rollback plan, you minimize downtime and ensure data reliability.

Engineering counterpart**3:18 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.