



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: qwen/qwen-2.5-7b-instruct

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

Duration: 32s

Actual run cost: \$0.000880

Projected execution cost: \$0.4840/mo (25/biz day × 22 days)

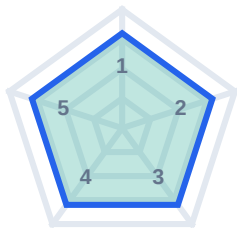
Report date: July 14, 2026 - 3:27 AM EST

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: 10989 chars - Latency: 32s - 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: 32s

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

Engineering counterpart

7:26 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 4s

1) Root Cause

The root cause is the implicit CROSS JOIN between `orders` (`o`), `order\_items` (`oi`), and `customers` (`c`). Without proper join conditions, Snowflake is performing a Cartesian product, which leads to an extremely large intermediate result set, causing significant performance issues.

2) Rewritten SQL with Explicit JOIN Predicates and Selective Columns

```
``sql
-- Optimized query
SELECT o.order_id, o.order_date, oi.product_id, 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

The optimization involves explicitly defining the join conditions between `orders` (`o`), `order\_items` (`oi`), and `customers` (`c`). By doing this, we eliminate the Cartesian product and reduce the intermediate data size, leading to more efficient processing. The selective columns in the `SELECT` clause further minimize the amount of data transferred and processed by the query. This results in a significant reduction in runtime, making the dashboard timeout issue a thing of the past.

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 10s

Sure, let's walk through the reasoning step-by-step and discuss the assumptions made.

Step-by-Step Reasoning and Assumptions

1. Identify Join Conditions:

- Assumption: The tables `orders` (`o`), `order\_items` (`oi`), and `customers` (`c`) are related through a common key.
- Assumption: The common key is `customer\_id` for `orders` and `customers`, and `order\_id` for `orders` and `order\_items`.

2. Explicit JOIN Predicates:

- Assumption: To ensure data integrity and to avoid Cartesian products, we need to specify join conditions.
- Explicit JOIN:

```
```sql
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN customers c ON o.customer_id = c.customer_id
```
```

3. Selective Columns:

- Assumption: We only need specific columns from each table for the query.
- Optimized SELECT:

```
```sql
SELECT o.order_id, o.order_date, oi.product_id, c.customer_id, c.region
```
```

4. WHERE Clause:

- Assumptions:
 - `o.order\_date >= '2024-01-01'` filters the orders to only include those from the specified date.
 - `c.region = 'EMEA'` filters the customers to only those in the EMEA region.
- Optimized WHERE:

```
```sql
WHERE o.order_date >= '2024-01-01'
 AND c.region = 'EMEA'
```
```

Summary of Optimizations

1. Explicit JOIN Predicates:

- Implicit CROSS JOIN: The original query performs a Cartesian product without join conditions, leading to a very large intermediate result set.
- Explicit JOIN: By specifying join conditions (`o.order\_id = oi.order\_id` and `o.customer\_id = c.customer\_id`), we eliminate the Cartesian product and ensure that only relevant rows are joined together.

2. Selective Columns:

- SELECT : Using `SELECT` can lead to transferring unnecessary data, which can be inefficient.
- Selective Columns: By specifying only the necessary columns (`o.order\_id`, `o.order\_date`, `oi.product\_id`, `c.customer\_id`, `c.region`), we reduce the amount of data that needs to be processed and transferred.

Assumptions

- Common Keys: Assuming that `orders` is linked to `order\_items` via `order\_id` and to `customers` via `customer\_id`.
- Data Integrity: Ensuring that the join conditions are correct and appropriately linking the tables.
- Filtering: Assuming that the date filter and region filter will significantly reduce the number of rows processed.

Final Optimized Query

```
```sql
-- Optimized query
SELECT o.order_id, o.order_date, oi.product_id, 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';
```
```

This optimized query should significantly reduce the runtime and improve the performance of the dashboard.

Engineering counterpart
7:26 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

7:26 AM - Responded in 10s

Before shipping the optimized query to production, it's crucial to verify several aspects to ensure that the changes will indeed improve performance and that the new query behaves as expected. Here are the steps you should take:

1. Initial Verification on a Subset of Data

- Purpose: Ensure the query works correctly and produces the expected results on a small subset of data.
- Action: Run the optimized query on a limited dataset (e.g., a few days of data) and compare the results with the original query.

2. Performance Metrics

- Purpose: Measure the performance of the optimized query.
- Action: Use Snowflake's performance metrics and query plans to compare the performance of the original and optimized queries.
- Metrics to Check:
 - Execution Time
 - Resource Consumption (CPU, IO, Memory)
 - Rows Processed
 - Intermediate Rows

3. Query Plan Comparison

- Purpose: Ensure the query plan is more efficient.
- Action: Use the `EXPLAIN` command to compare the execution plans of both queries.
- Expected Outcome: The optimized query should have a more efficient plan, with fewer intermediate rows and reduced resource consumption.

4. Stress Testing

- Purpose: Ensure the optimized query handles large datasets efficiently.
- Action: Run the optimized query on larger datasets to simulate production conditions.
- Expected Outcome: Confirm that the query performs well without significant degradation in performance.

5. Error Handling and Stability

- Purpose: Ensure the query is robust and stable.
- Action: Test the query under various edge cases and error scenarios (e.g., no matching customers or orders).
- Expected Outcome: The query should handle edge cases gracefully without errors.

6. User Feedback

- Purpose: Gather feedback from users to ensure they experience the improved performance.
- Action: Deploy the optimized query temporarily and gather user feedback, especially from those who were experiencing timeouts.
- Expected Outcome: Users should report faster response times and no new issues.

7. Logging and Monitoring

- Purpose: Ensure ongoing monitoring for any new issues.
- Action: Set up logging and monitoring for the query to catch any performance issues or unexpected behavior.
- Expected Outcome: Continuous monitoring should show stable and improved performance.

Example Verification Steps

Step 1: Initial Verification on a Subset of Data

```
```sql
-- Limited dataset verification
SELECT o.order_id, o.order_date, oi.product_id, 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'
AND o.order_id > '2024-01-01-0001'; -- Limit to a subset of data
'''
```

#### #### Step 2: Performance Metrics

```
```sql
-- Original query performance
USE WAREHOUSE <your_warehouse>;
EXPLAIN SELECT FROM orders o, order_items oi, customers c WHERE o.order_date >= '2024-01-01' AND c.region =
'EMEA';

-- Optimized query performance
EXPLAIN SELECT o.order_id, o.order_date, oi.product_id, 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';
'''
```

Step 3: Query Plan Comparison

```
```sql
-- Original
```

**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 9s

### ### Top Risks and Rollback Plan

#### #### Top Risks

##### 1. Incorrect Join Conditions:

- Risk: If the join conditions are incorrect, the query could return incorrect results or no results at all.
- Mitigation: Verify join conditions meticulously and test on a small subset of data.

##### 2. Performance Degradation:

- Risk: The optimized query might perform worse than the original query, especially if the assumptions about data distribution are incorrect.
- Mitigation: Measure performance metrics and compare them against the original query.

##### 3. Resource Overload:

- Risk: The optimized query might consume more resources, leading to performance degradation for other processes.
- Mitigation: Monitor resource consumption and ensure the query does not overload the system.

##### 4. Data Consistency:

- Risk: If the optimized query introduces data inconsistency, it could cause issues in the reporting or analytics.
- Mitigation: Thoroughly test the query with different subsets of data and ensure data consistency.

##### 5. User Experience:

- Risk: Users might experience unexpected delays or errors due to changes in the query.
- Mitigation: Gather user feedback and monitor user interactions.

#### #### Rollback Plan

##### 1. Backup Current Query:

- Step: Ensure the original query is backed up and easily accessible.
- Action: Save the original query and its execution plan for reference.

##### 2. Revert Query:

- Step: Revert the query to its original form if issues are detected.
- Action: Simply execute the original query and verify that it works as expected.

##### 3. Monitor Post-Deployment:

- Step: Continuously monitor the system for any issues after deployment.
- Action: Set up alerting and monitoring to catch any anomalies quickly.

##### 4. Rollback Procedure:

- Step: Develop a clear rollback procedure.
- Action: Write down the steps to revert the query and ensure the rollback is tested in a staging environment.

### ### Example Rollback Plan

##### 1. Backup Original Query:

```
```sql
-- Save original query
CREATE OR REPLACE VIEW original_query AS SELECT  FROM orders o, order_items oi, customers c WHERE
o.order_date >= '2024-01-01' AND c.region = 'EMEA';
```
```

##### 2. Revert Query:

```
```sql
-- Revert to original query
CREATE OR REPLACE VIEW optimized_query AS SELECT  FROM orders o, order_items oi, customers c WHERE
o.order_date >= '2024-01-01' AND c.region = 'EMEA';
```
```

### 3. Monitor Post-Deployment:

- Action: Use Snowflake's monitoring tools to track resource usage, query performance, and user interactions.
- Alerts: Set up alerts for any unexpected performance drops or errors.

### 4. Rollback Procedure:

- Steps:
  1. Stop the current query.
  2. Drop the view created with the optimized query.
  3. Restore the original query view.
  4. Verify the original query works as expected.

## ### Final Steps

### 1. Deploy Optimized Query:

```
```sql
-- Deploy optimized query
CREATE OR REPLACE VIEW optimized_query AS
SELECT o.order_id, o.order_date, oi.product_id, 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';
```
```

### 2. Test and Monitor

**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.