



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: google/gemini-2.5-flash

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

Duration: 21s

Actual run cost: \$0.005300

Projected execution cost: \$2.92/mo (25/biz day × 22 days)

Report date: July 13, 2026 - 11:16 PM 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: 11623 chars - Latency: 21s - 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: 21s

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

Engineering counterpart

3:16 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 5s

Here's the analysis and rewritten query:

1) Root Cause

The primary root cause is an implicit `CROSS JOIN` between `orders`, `order\_items`, and `customers` tables due to missing join predicates in the `WHERE` clause. This generates an enormous intermediate result set (approaching the product of all filtered rows) before any relevant joins can be performed, leading to excessive data processing and I/O.

2) Rewritten SQL

```
```sql
SELECT
 o.order_id,
 o.order_date,
 o.customer_id,
 oi.product_id,
 oi.quantity,
 oi.price,
 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) Brief Performance Rationale

1. Explicit JOINS: Explicit `JOIN` clauses (`INNER JOIN ... ON`) clearly define the relationships between tables. This allows the query optimizer to efficiently combine rows based on matching keys (`o.order\_id = oi.order\_id` and `o.customer\_id = c.customer\_id`), avoiding the costly implicit `CROSS JOIN`.
2. Selective Columns: Replacing `SELECT` with specific column names (`o.order\_id, o.order\_date, ...`) reduces the amount of data transferred and processed from disk to memory, and across network stages, further optimizing I/O and memory usage.
3. Predicate Pushdown: By defining proper joins, the filters (`o.order\_date >= '2024-01-01'` and `c.region = 'EMEA'`) can be applied earlier in the execution plan, significantly reducing the number of rows processed in subsequent join operations. This dramatically shrinks the intermediate result set size compared to the original query where the cross-product is computed first.

Engineering counterpart

3:16 AM - Responded in 0s

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

Agent response

3:16 AM - Responded in 5s

Let's break down the reasoning step-by-step, including the assumptions made.

Step 1: Analyze the Original Query and `EXPLAIN` Plan

Original Query:

```
```sql
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
```
```

Reasoning from Analysis:

1. Old-Style `FROM` Clause: The `FROM orders o, order\_items oi, customers c` syntax is characteristic of older SQL standards for defining joins. When no `ON` clause or join condition is specified, this defaults to a `CROSS JOIN`.
2. Missing Join Predicates: The critical piece of information from the `EXPLAIN` note and the query itself is "no join predicates between o, oi, c." This directly confirms the implicit `CROSS JOIN`.
3. Magnitude of `CROSS JOIN`:
 - `ORDERS` after filtering will have `12,400,000` rows.
 - `CUSTOMERS` after filtering will have `890,000` rows.
 - `ORDER\_ITEMS` has `452,000,000` rows (unfiltered by the `WHERE` clause).
 - A `CROSS JOIN` of `ORDERS` and `CUSTOMERS` alone would produce $12.4M \times 890K = 11 \text{ trillion}$ rows.
 - Then, `CROSS JOINING` with `ORDER\_ITEMS` would multiply this by `452M`. The resulting intermediate dataset would be astronomically large, easily exceeding any practical memory or disk capacity, leading to extreme slowness and likely timeouts. This fully explains the "avg runtime 847s, job id JOB-88421, Symptom: dashboard timeout."
4. `SELECT` Usage: While not the primary cause of the timeout, `SELECT` is generally considered bad practice in production queries because it retrieves all columns, even if they're not needed. This can increase network traffic, memory consumption, and disk I/O.

Step 2: Formulate Assumptions about Table Relationships

Since the original query intended to join these tables (otherwise, why select from all three for a sales rollup?), I need to assume standard relational database patterns for `orders`, `order\_items`, and `customers`.

Assumptions Made:

1. `orders` and `customers`: An `orders` table typically has a foreign key referencing the `customers` table.
Assumption: `orders.customer\_id` is the foreign key linking to `customers.customer\_id`.
2. `orders` and `order\_items`: An `orders` table typically has a one-to-many relationship with an `order\_items` table, where `order\_items` stores the individual products within an order.
Assumption: `order\_items.order\_id` is the foreign key linking to `orders`.

What would you verify before shipping this to production?

Agent response

3:16 AM - Responded in 6s

Before shipping this optimized query to production, I would verify several key aspects to ensure its correctness, performance, and impact.

Here's a checklist of verifications:

1. Functional Correctness (Does it return the right data?):

Data Integrity: Compare a sample of the results from the optimized query with a sample from the original query (if the original query ever completed for a small enough dataset). This is crucial to ensure the join conditions I assumed (e.g., `o.customer_id = c.customer_id`) are indeed the correct ones from the data model perspective. Pay close attention to row counts and aggregated values if applicable.

Edge Cases: Test with `o.order_date` near the boundary (`'2024-01-01'`), and with regions other than 'EMEA' (to ensure filters work as expected).

Nulls: Understand how `NULL` values in join keys (e.g., `o.customer_id` being `NULL`) might affect the `INNER JOIN` vs. the implicit `CROSS JOIN`. An `INNER JOIN` will exclude rows with `NULL`'s in the join key, while a `CROSS JOIN` followed by a filter might behave differently. Assuming standard data integrity, this usually isn't an issue for FKs.

2. Performance Verification (Is it fast enough?):

EXPLAIN Plan Review: Run `EXPLAIN` on the optimized query. I'd expect to see:

Explicit `JOIN` operations (e.g., `JOIN HASH` or `JOIN MERGE`).

Predicate pushdown: The `WHERE` clause filters (`order_date >= '2024-01-01'` and `region = 'EMEA'`) should be applied early, ideally before or during table scans, significantly reducing the number of rows processed by subsequent joins.

Reduced row estimates: The estimated number of rows for intermediate steps should be drastically lower than the original `EXPLAIN` plan.

No `CROSS JOIN` operations.

Execution Time & Resource Usage:

Run the optimized query on a representative dataset (e.g., a clone of the production environment, or a sufficiently large dev/staging environment).

Measure actual execution time. Is it within the acceptable SLA for the dashboard (e.g., < 30 seconds, < 60 seconds)?

Monitor Snowflake's query profile for the optimized query:

Spill to Disk: Ensure there's minimal to no spilling to local disk or remote disk, indicating sufficient memory allocation for the chosen warehouse size.

Pruning: Check if micro-partitions are being effectively pruned for `order_date` and `region` filters.

Scanning: Observe the amount of data scanned from each table to ensure it aligns with expectations after filters.

Warehouse Size: Confirm the current virtual warehouse size is appropriate for the query's resource needs. If the query is still slow, a larger warehouse might be temporarily needed, or further indexing/clustering considered.

Concurrency: Consider if this query will run frequently or in parallel with other heavy queries. Will it cause warehouse contention?

3. Data Model & Metadata Review:

Confirm Primary/Foreign Keys: Explicitly verify the actual primary/foreign key relationships in the Snowflake schema for `orders`, `order_items`, and `customers`. My assumptions (`o.customer_id = c.customer_id`, `o.order_id = oi.order_id`) must

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

Agent response

3:16 AM - Responded in 6s

Given the drastic nature of the original problem (implicit `CROSS JOIN`), the risks associated with this specific optimization are relatively low if the assumed join conditions are correct. However, it's always good practice to consider potential issues.

Top Risks

1. Incorrect Join Logic (Highest Risk if Assumptions are Wrong):

Description: The biggest risk is that my assumed join conditions (`o.customer\_id = c.customer\_id` and `o.order\_id = oi.order\_id`) are not the actual or intended relationships in the database schema or the business logic. If the `customer\_id` column in `orders` doesn't link to the `customer\_id` in `customers`, or if `order\_id` in `order\_items` doesn't link to `orders`, the query will return incorrect results, or results with an unintended scope.

Impact: The dashboard (EMEA regional sales rollup) will display incorrect sales figures, customer counts, or order details. This could lead to bad business decisions, financial discrepancies, or customer service issues. This is the most critical risk for data integrity.

Mitigation: This is primarily mitigated by the "Functional Correctness (Data Integrity)" verification step (comparing sample results, schema review, and asking the business user/data modeler).

2. Performance Degradation on Edge Cases / Unexpected Data Distribution (Low Risk for THIS query, but generally applicable):

Description: While the rewritten query is fundamentally more efficient, there's a slight theoretical risk that for very specific, unforeseen data distributions or highly skewed data, the chosen join algorithm by Snowflake (e.g., hash join vs. sort-merge join) might perform worse than an alternative, or that specific filters become less effective than anticipated. For this particular type of fix, this risk is very low because the original problem was so severe.

Impact: The query might still be slow, although almost certainly faster than the original. It might not meet the required SLA for the dashboard.

Mitigation: Thorough "Performance Verification" (EXPLAIN plan, actual run time, query profile analysis on representative data) handles this.

3. Resource Contention / Increased Warehouse Load (Moderate Risk):

Description: While the optimized query itself uses resources more efficiently, if it's now run much more frequently (because it's fast) or if the virtual warehouse is undersized for the new workload, it could lead to increased load on the warehouse, potentially impacting other queries running on the same warehouse.

Impact: Other dashboards or data processes could slow down or experience delays.

Mitigation: Monitor warehouse usage metrics (CPU, memory, query queue time) after deployment. If contention is observed, consider scaling up the warehouse or adjusting its auto-suspend/resume settings. For a critical dashboard, a dedicated (even if small) warehouse might be considered.

Rollback Plan

The rollback plan for a query optimization is typically straightforward because you're not changing any underlying data or schema, only the definition of a view or the query itself.

1. Revert to the Previous Query Version:

For Ad-hoc/Direct Query: If the change was just an ad-hoc query directly run by a user, the rollback is simply to stop using the new query and revert to the old one (if it was ever used successfully for small datasets, or tolerated for its slowness).

For Dashboard/Application: If this query powers a dashboard or application:

Identify Source: Locate where the SQL is stored/defined (e.g., BI

Engineering counterpart**3:16 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.