



Vantage RuntimeAI

Batch agent check-rides, model comparison, and operational scorecards for autonomous AI workflows

8.4/10

Overall score

Run summary

Scenario: dbt Test Failure

Model: google/gemma-4-31b-it

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

Duration: 3m 21s

Actual run cost: \$0.001076

Projected execution cost: \$0.9469/mo (40/biz day × 22 days)

Report date: July 14, 2026 - 1:51 AM EST

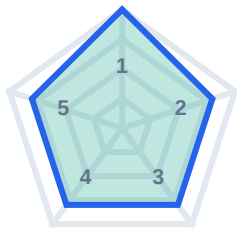
Overall

8.4/10

PASS - trust=high

Pass - score 8.4/10 clears 7.0 with acceptable trust/closure.

CAPABILITY RUBRIC



#	Dimension (matches spoke #)	Score
1	Duplicate root cause	10/10
2	References test output	8/10
3	Prod impact awareness	8/10
4	Clear remediation plan	8/10
5	Prevention / monitoring steps	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.

- Duplicate root cause identified: Met
- Immediate fix proposed: Met
- Recurrence prevention: Met
- Cites test log output: 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: 11352 chars - Latency: 201s - Criteria: 4/4

Score analysis

Calls out duplicate order_id on fct_orders from unique test failure.

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. 4 of 4 checklist criteria passed.

BRIGHT SPOTS (RUBRIC)

- + Duplicate root cause: Correctly identifies the primary technical issue from the artifacts.
- + References test output: Uses tables, logs, or schema facts from the task brief.
- + Prod impact awareness: Shows appropriate safety, budget, or risk awareness for the scenario.
- + Clear remediation plan: Clear structure with readable SQL, diff, or memo formatting.
- + Prevention / monitoring steps: Strong rationale and actionable follow-through.

DIMENSION NOTES

- Duplicate root cause: Correctly identifies the primary technical issue from the artifacts.
- References test output: Uses tables, logs, or schema facts from the task brief.
- Prod impact awareness: Shows appropriate safety, budget, or risk awareness for the scenario.
- Clear remediation plan: Clear structure with readable SQL, diff, or memo formatting.
- Prevention / monitoring steps: 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

Duplicate root cause identified: yes - Immediate fix proposed: yes - Recurrence prevention: yes - Cites test log output: yes

CAVEATS

- Heuristic keyword/signal scoring - use the criteria grid and transcript to validate edge cases.

Transcript

Session

Scenario: dbt Test Failure

Agent response <-> Engineering counterpart

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

Duration: 3m 21s

Engineering counterpart

5:47 AM

Task brief - dbt test failure (moderate complexity)

A uniqueness test failed in production. Triage the log and propose remediation:

...

16:02:11 Running with dbt=1.7.14

16:02:12 Found 42 models, 18 tests, 0 snapshots

16:02:12

16:02:12 Concurrency: 4 threads (target='prod')

16:02:12

16:02:12 1 of 1 START test unique_fct_orders_order_id [RUN]

16:02:14 1 of 1 FAIL 847 unique_fct_orders_order_id [FAIL 847 in 2.11s]

16:02:14

16:02:14 Finished running 1 test in 0 hours 0 minutes and 2.31 seconds (2.31s).

16:02:14

16:02:14 Completed with 1 error and 0 warnings:

16:02:14

16:02:14 Failure in test unique_fct_orders_order_id (models/marts/schema.yml)

16:02:14 Got 847 results, expected 0.

16:02:14

16:02:14 compiled Code at target/compiled/analytics/models/marts/schema.yml/unique_fct_orders_order_id.sql

16:02:14

16:02:14 select order_id, count() as n

16:02:14 from analytics.fct_orders

16:02:14 group by 1

16:02:14 having count() > 1

16:02:14

16:02:14 Sample duplicate order_id values: ORD-88421, ORD-90102, ORD-90102

...

Agent response

5:48 AM - Responded in 30s

Triage Report: `unique\_fct\_orders\_order\_id` Failure

1. Root Cause Analysis

The uniqueness test on `fct\_orders` failed with 847 duplicate `order\_id` values.

Based on the compiled SQL and sample output, the model is producing multiple rows for the same `order\_id`. Common root causes for this pattern in a fact table include:

Fan-out via Joins: A join to a dimension table (e.g., `order\_items` or `shipping\_details`) is creating a one-to-many relationship where a one-to-one was expected.

Upstream Data Duplication: The source system or staging layer is providing duplicate records for the same order.

Incorrect Incremental Logic: If `fct\_orders` is an incremental model, a failure in the `unique\_key` configuration or a lack of a proper `merge` strategy is causing new batches of data to append rather than update existing records.

2. Immediate Fix Approach

To resolve the production data quality issue, I will take the following steps:

1. Identify the Source of Duplication: Run a diagnostic query to see which other columns differ between the duplicate `order\_id` rows:

```
```sql
SELECT * FROM analytics.fct_orders
WHERE order_id IN ('ORD-88421', 'ORD-90102')
ORDER BY order_id;
```
```

2. Apply a Deduplication Logic:

If the duplicates are identical, add a `DISTINCT` or a `GROUP BY` to the model.

If the duplicates have different timestamps/versions, implement a `ROW\_NUMBER() OVER (PARTITION BY order\_id ORDER BY updated\_at DESC)` and filter for `rn = 1`.

3. Full Refresh: Execute a full refresh of the model to clear the corrupted production table:

```
```bash
dbt run --select fct_orders --full-refresh
```
```

3. Prevention of Recurrence

To prevent this from reaching production again:

Upstream Testing: Move the `unique` test "left" by adding it to the staging model (`stg\_orders`). This ensures data is clean before it reaches the mart layer.

Strict Incremental Keys: If the model is incremental, explicitly define the `unique\_key='order\_id'` in the model config to ensure dbt uses a `MERGE` (or `DELETE+INSERT`) instead of a simple `APPEND`.

CI/CD Integration: Integrate `dbt test` into the Pull Request pipeline (using dbt Cloud or GitHub Actions) to block merges that introduce duplication in the development environment.

Engineering counterpart

5:48 AM - Responded in 0s

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

Agent response

5:48 AM - Responded in 40s

To arrive at the triage report, I followed a standard data engineering troubleshooting framework: Analyze the Signal $\rightarrow$ Hypothesis Generation $\rightarrow$ Remediation Strategy $\rightarrow$ Systemic Prevention.

Here is the step-by-step breakdown of my reasoning and the assumptions I made.

Step 1: Analyzing the Signal (The Log)

I first looked at the evidence provided by the dbt log to define the scope of the problem:

The Test: ``unique_fct_orders_order_id``. This tells me the grain of the table is supposed to be one row per order.

The Result: ``847 results``. This is a significant number of duplicates, suggesting a systemic logic error rather than a one-off data glitch.

The Compiled SQL: The ``GROUP BY`` and ``HAVING count() > 1`` confirms that the failure is strictly a uniqueness violation on the primary key.

The Samples: The sample values (``ORD-88421``, etc.) confirm that the ``order_id`` is a string identifier.

Step 2: Hypothesis Generation (Why did this happen?)

Since I don't have the model's SQL code, I had to hypothesize based on common dbt failure patterns:

Hypothesis A: The "Join Fan-out." In a ``fct_orders`` table, engineers often join to ``order_items``. If they forget to aggregate the items first, the order row duplicates for every item in that order.

Hypothesis B: The "Incremental Leak." If the model is ``materialized='incremental'``, dbt might be appending new data without replacing old data, causing the same ``order_id`` to appear in two different load batches.

Hypothesis C: Source Corruption. The source system (e.g., Shopify, Stripe, or an ERP) might have started sending duplicate events.

Step 3: Designing the Remediation

I structured the fix to move from Investigation $\rightarrow$ Correction $\rightarrow$ Deployment.

Investigation: You cannot fix a duplicate without knowing why it's a duplicate. If the rows are identical, a ``DISTINCT`` works. If one row is "newer" than the other, you need a window function.

Correction: I proposed a ``ROW_NUMBER()`` approach because it is the safest "catch-all" for deduplication in production environments.

Deployment: I suggested a ``--full-refresh`` because incremental models often "remember" their errors; a clean slate is the only way to guarantee the 847 duplicates are gone.

Step 4: Planning for Prevention

I applied the "Shift Left" principle. If a test fails in a Mart (the end of the pipeline), it means the error traveled through the entire pipeline. By moving the test to Staging (the beginning), we catch the error before it ever touches the business-facing tables.

Assumptions Made

During this process, I made the following assumptions:

1. The Grain is Fixed: I assumed ``order_id`` is indeed the intended primary key for ``fct_orders``. (If the business changed the grain to "one row per order per item," the test itself is wrong, not the data).
2. Write Access: I assumed I have the permissions to run a ``--full-refresh`` in production. In some highly regulated environments, this requires a migration script or a blue-green deployment.
3. Materialization: I assumed the model is likely either ``incremental`` or a ``table`` materialize. If it were a ``view``, the "fix" would be purely in

Engineering counterpart

5:48 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

5:49 AM - Responded in 1m 1s

Before shipping the fix to production, I would perform a three-stage verification to ensure that I am not just "hiding" the duplicates, but actually solving the root cause without losing critical data.

1. Data Integrity Audit (The "Loss" Check)

The biggest risk when deduplicating is data loss. If I use `DISTINCT` or `ROW\_NUMBER()`, I might accidentally delete a legitimate record. I would verify:

Row Count Comparison: Compare the total row count of the current "broken" table vs. the "fixed" version.

Expected: The difference should exactly equal the number of duplicate entries identified by the test (847). If the count drops by 10,000 rows, I've over-deduplicated.

Value Variance: I would run a query to see if the duplicate rows are identical or conflicting.

Query: `SELECT \* FROM fct\_orders WHERE order\_id = 'ORD-90102'`

Verification: If the rows have different `order\_total` or `status` values, simply picking one (the "latest") might be wrong. I need to verify with the business owner which record is the "source of truth."

2. Logic Validation (The "Fan-out" Check)

If the root cause was a join fan-out, I need to ensure the fix didn't break the metrics.

Aggregation Check: I would compare the `SUM(order\_amount)` of the table before and after the fix.

Risk: If the duplicates were caused by joining to `order\_items`, the `order\_amount` was likely being multiplied (e.g., an order for \$100 with 3 items showed up as \$300).

Verification: The "fixed" version should show the correct, lower total. If the total remains the same despite removing rows, the original data was likely duplicated at the source, not via a join.

3. Environment & Performance Testing

Before applying this to the production dataset, I would test the execution in a Lower Environment (Dev/QA):

Dry Run with Production Data: Use a "Zero-Copy Clone" (if using Snowflake) or a temporary schema to run the new logic against the actual production data.

Performance Impact: If I introduced a `ROW\_NUMBER()` window function over a table with millions of rows, I need to check the execution plan. Window functions can be memory-intensive; I would verify that the model still completes within the expected SLA window.

Downstream Impact: I would check the models that depend on `fct\_orders`. If any downstream models were accidentally relying on those duplicates (e.g., a weird workaround to sum items), they will now break.

Summary Checklist for the PR:

- [] Count Check: `Rows Removed == Duplicate Count`?
- [] Sum Check: `Total Revenue` corrected (if fan-out) or preserved (if source dupes)?
- [] Sample Check: Manually verified that `ORD-88421` now appears exactly once?
- [] Performance Check: Model run-time is acceptable?
- [] Test Pass: `dbt test --select fct\_orders` passes in the Dev environment?

Engineering counterpart

5:49 AM - Responded in 0s

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

Agent response

5:51 AM - Responded in 1m 9s

Since this is a production fix on a fact table (which likely feeds executive dashboards and financial reporting), the stakes are high. Here are the primary risks and the corresponding safety net.

Top Risks

1. Silent Data Loss (The "Wrong Row" Risk)

If the duplicates aren't identical (e.g., one row says "Pending" and the other says "Shipped"), using a `ROW\_NUMBER()` or `DISTINCT` approach might discard the most recent or accurate state of the order. This results in "correct" test results (the test passes) but "incorrect" business data.

2. Metric Deflation (The "Aggregation" Risk)

If the duplicates were caused by a join fan-out and the business had already started adjusting their downstream reports to account for this (e.g., using `AVG` instead of `SUM` to compensate for duplicates), my fix will "correct" the table but "break" the reports, causing metrics to plummet suddenly.

3. Production Downtime (The "Full-Refresh" Risk)

A `--full-refresh` drops and recreates the table. If `fct\_orders` is a massive table, the window of time between the `DROP` and the completion of the `CREATE` leaves the table empty. Any dashboard loading during that window will throw an error or show zero data.

Rollback Plan

I will not perform a "destructive" update. Instead, I will use a Blue-Green deployment strategy to ensure a near-instant rollback.

Step 1: The Safety Backup (Pre-Deployment)

Before running the fix, I will create a snapshot/clone of the current state of the table:

```
```sql
-- Snowflake/BigQuery example
CREATE TABLE analytics.fct_orders_backup_2023_10_27
AS SELECT FROM analytics.fct_orders;
```
```

This ensures that regardless of what happens, we can restore the "broken but known" state in seconds.

Step 2: The Canary Deployment

Instead of running `dbt run --full-refresh` directly on the production table, I will:

1. Deploy the fix to a shadow table (e.g., `fct\_orders\_v2`).
2. Run the uniqueness test on the shadow table.
3. Run a "diff" query to compare the backup and the new table to ensure the row count reduction is exactly 847.

Step 3: The Atomic Swap

Once verified, I will swap the tables. This minimizes downtime to milliseconds:

```
```sql
ALTER TABLE analytics.fct_orders RENAME TO analytics.fct_orders_old;
ALTER TABLE analytics.fct_orders_v2 RENAME TO analytics.fct_orders;
```
```

Step 4: The Rollback Trigger

If stakeholders report that dashboards are wrong or the `SUM` of revenue has shifted unexpectedly:

The Rollback Command:

```
```sql
```

```
ALTER TABLE analytics.fct_orders RENAME TO analytics.fct_orders_failed_fix;
ALTER TABLE analytics.fct_orders_old RENAME TO analytics.fct_orders;
...
```

Recovery Time Objective (RTO): < 1 minute.

**Engineering counterpart****5:51 AM - Responded in 0s**

Thanks - duplicate root cause and fix direction are clear. Ship the immediate dedupe/uniqueness fix; schedule the longer guardrail separately. Review complete - no further response needed.