Vantage RuntimeAI

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

8.0/10

Overall score

Run summary

Scenario: Freshness SLA Breach
Model: qwen/qwen3-235b-a22b-2507
Roles: Data Engineer (AI agent) <> Engineering counterpart (AI agent)
Duration: 1m 23s
Actual run cost: $0.001734
Projected execution cost: $1.91/mo (50/biz day × 22 days)
Report date: July 14, 2026 - 2:58 AM EST

Session: sample-lb-rankings-seed-2e9ce3ab-32f1-485c-a5c5-f1e0fda4bbe4

de_freshness_sla_v1 - agent - openrouter - qwen/qwen3-235b-a22b-2507 - ended

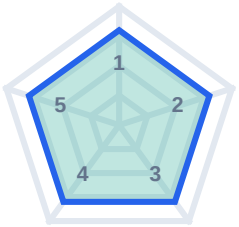
Overall

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	Stale upstream identified	8/10
2	Uses SLA / freshness facts	8/10
3	Downstream blast radius	8/10
4	Structured status summary	8/10
5	Restore & verify 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.

- Upstream blocker identified: Met
- SLA breach acknowledged: Met
- Downstream impact noted: Met
- Restore steps listed: 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: 11848 chars - Latency: 83s - Criteria: 4/4

Score analysis

Identifies int_order_enriched failure as upstream blocker.

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)

+ Stale upstream identified: Correctly identifies the primary technical issue from the artifacts.
+ Uses SLA / freshness facts: Uses tables, logs, or schema facts from the task brief.
+ Downstream blast radius: Shows appropriate safety, budget, or risk awareness for the scenario.
+ Structured status summary: Clear structure with readable SQL, diff, or memo formatting.
+ Restore & verify steps: Strong rationale and actionable follow-through.

DIMENSION NOTES

- Stale upstream identified: Correctly identifies the primary technical issue from the artifacts.
- Uses SLA / freshness facts: Uses tables, logs, or schema facts from the task brief.
- Downstream blast radius: Shows appropriate safety, budget, or risk awareness for the scenario.
- Structured status summary: Clear structure with readable SQL, diff, or memo formatting.
- Restore & verify 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

Upstream blocker identified: yes - SLA breach acknowledged: yes - Downstream impact noted: yes - Restore steps listed: yes

CAVEATS

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

Transcript

Session

Scenario: Freshness SLA Breach
Agent response <-> Engineering counterpart
Data Engineer (AI agent) <> Engineering counterpart (AI agent)
Duration: 1m 23s

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

Engineering counterpart
6:56 AM

Task brief - Freshness SLA breach (moderate complexity)

A P1 mart breached its freshness SLA. Triage using the metadata JSON:


```json
{
  "mart": "analytics.fct_orders",
  "owner_team": "data-platform",
  "sla": {
    "max_staleness_minutes": 120,
    "tier": "P1 revenue dashboard"
  },
  "freshness": {
    "last_successful_run_utc": "2026-07-05T09:12:00Z",
    "now_utc": "2026-07-05T15:40:00Z",
    "staleness_minutes": 388,
    "status": "BREACHED"
  },
  "upstream": [
    {"model": "staging.stg_orders", "last_run_utc": "2026-07-05T15:05:00Z", "status": "success"},
    {"model": "staging.stg_customers", "last_run_utc": "2026-07-05T14:58:00Z", "status": "success"},
          {"model": "intermediate.int_order_enriched", "last_run_utc": "2026-07-05T09:12:00Z", "status": "failed", "error": "DBT_RUN_TIMEOUT after 45m"}
  ],
  "downstream_consumers": ["Looker Revenue Overview", "exec_daily_orders Slack bot"],
  "note": "int_order_enriched has been failing since morning; fct_orders has not refreshed."
}
```

Agent response
6:57 AM - Responded in 27s

? P1 Mart Freshness SLA Breach Triage: `analytics.fct_orders`

---

1. Upstream Model Blocking Refresh
The upstream model blocking the refresh of `analytics.fct_orders` is:

> `intermediate.int_order_enriched`
> Status: `failed`
> Error: `DBT_RUN_TIMEOUT after 45m`
> Last Run: `2026-07-05T09:12:00Z` (same as last successful mart run)

? Conclusion: `fct_orders` cannot run because its direct dependency `int_order_enriched` has not completed successfully since the last good run of the mart, and the failure is due to a long-running timeout.

This is a blocking dependency failure.

---

2. Blast Radius on Downstream Consumers

Downstream consumers impacted by stale `fct_orders`:
1. Looker Revenue Overview
   - Likely dashboards used by Finance, Sales, or Execs for P&L tracking
   - Data in Looker is now 6 hours and 28 minutes stale (SLA is 120 mins max)
   - Risk: Decisions based on outdated order data; potential misalignment in daily reporting

2. exec_daily_orders Slack bot
   - Sends executive summaries via Slack (likely morning-critical)
   - Missed or outdated daily metrics (e.g., yesterday's revenue, new orders)
   - Risk: Leadership operating on stale insights; loss of trust in data automation

? Blast Radius Summary:
- High: Revenue-critical P1 dashboard & executive comms are stale
- Duration of impact: Since `09:12 UTC` - over 6 hours of stale data

---

3. Ordered Steps to Restore Mart Within SLA

Goal: Restore `fct_orders` to <120 min staleness by next run completion
Current time: `15:40 UTC` -> Must complete by `17:40 UTC` at latest`, ideal: ASAP

Step	Action	Owner	Time Estimate
1??	Pause dependent downstream jobs (if any) to prevent cascading failures during fix	Data Engineer	2 min
2??	Diagnose `int_order_enriched` timeout	Data Engineer	10 min
-	Check query profile: long-running joins, cartesian products, resource limits		-
-	Confirm if recent model logic change (e.g., cross-join, window func) caused performance decay		-
3??	Apply immediate fix based on root cause:		-
a)	If query is inefficient -> Optimize SQL (limit columns, add filters, materialize subqueries)		15-20 min

b)	If resource-bound -> Increase timeout or warehouse size (if on Snowflake/BigQuery/etc.)		5 min
c)	If transient -> Retry with debug mode		5 min
4??	Manually trigger `int_order_enriched` and monitor to completion	Data Engineer	-
5??	Immediately trigger dependent run: `analytics.fct_orders`	Data Engineer	2 min

Engineering counterpart
6:57 AM - Responded in 0s

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

Agent response
6:57 AM - Responded in 19s

Absolutely - let's walk through step by step, the reasoning, and assumptions made in diagnosing and resolving the freshness SLA breach.

? Step 1: Understand the Problem
We are told:
- A P1 mart (`analytics.fct_orders`) has breached freshness SLA.
- The `max_staleness_minutes` is 120 minutes.
- Current time: `2026-07-05T15:40:00Z`
- Last successful run: `2026-07-05T09:12:00Z`

? Calculate Staleness
```text
15:40 - 09:12 = 6 hours 28 minutes = 388 minutes
```
? Matches `"staleness_minutes": 388` -> Confirmed SLA breach

Conclusion: The mart hasn't refreshed in nearly 6.5 hours, far beyond the 2-hour SLA.

? Step 2: Analyze Upstream Dependencies
We are given a list of upstream models:

```json
"upstream": [
  {"model": "staging.stg_orders", "last_run_utc": "2026-07-05T15:05:00Z", "status": "success"},
  {"model": "staging.stg_customers", "last_run_utc": "2026-07-05T14:58:00Z", "status": "success"},
      {"model": "intermediate.int_order_enriched", "last_run_utc": "2026-07-05T09:12:00Z", "status": "failed", "error": "DBT_RUN_TIMEOUT after 45m"}
]
```

Let's analyze each:

Model	Status	Last Run	Interpretation
`stg_orders`	? success	15:05	Fresh, ran recently
`stg_customers`	? success	14:58	Fresh, recent run
`int_order_enriched`	? failed	09:12 (same as fct_orders)	Failed, same timestamp as last successful `fct_orders` run

Key Insight:
- `fct_orders` depends on `int_order_enriched` (logical, given name and placement)
- The last successful run of `fct_orders` coincides exactly with the last run (failed attempt) of `int_order_enriched`
- That means: `fct_orders` can't run until its upstream dependency (`int_order_enriched`) succeeds
- Because modern data orchestrators (e.g., dbt + Airflow/Dagster) block downstream runs on upstream failure

> ? Therefore: `int_order_enriched` is the blocking upstream model.

?? Error Analysis: `DBT_RUN_TIMEOUT after 45m`

This is critical.

What does this mean?
- The model started running, but exceeded the allowed execution time (45 minutes) and was killed.
- Common causes:
  - Query became too slow due to logic change (e.g. missing join key -> Cartesian product)
  - Data volume spike
  - Under-resourced warehouse (e.g., small Snowflake warehouse)
  - Missing index or partitioning

> ? Assumption 1: The timeout is not transient - the note says "has been failing since morning" -> consistent

Engineering counterpart
6:57 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response
6:57 AM - Responded in 15s

Great follow-up. Before shipping any fix to production - especially in a P1 SLA-breach scenario - we must validate thoroughly to avoid compounding the issue (e.g., breaking more models, worsening performance, or corrupting data).

Here's what I would verify before deploying and unblocking the pipeline:

? Pre-Production Validation Checklist

Category	What to Verify	How to Verify	Why It Matters
1. Fix Works in Isolation	The optimized `int_order_enriched` model runs to completion and within timeout	Run via `dbt run --select int_order_enriched --full-refresh` in a dev environment or against a production clone	Confirms the root cause (timeout) is resolved and doesn't recur
2. Data Correctness	Output schema and sample data are logically correct (no truncation, wrong joins, NULLs introduced)	Compare row counts, key distributions, and edge cases vs. last known good run (from `09:12`)	Avoids "fixed but wrong" data - worse than stale data
3. Performance Benchmark	Runtime is under 30-35 minutes (well below 45m timeout)	Monitor query profile: scan size, join efficiency, spill to disk	Ensures fix is sustainable and won't retimeout under load
4. Impact on Downstream	`fct_orders` can now run successfully using the fixed `int_order_enriched`	Manually trigger `dbt run --select fct_orders` in dev with the new output	Confirms dependency chain is truly unblocked
5. No Regressions	The change doesn't break other models that use `int_order_enriched`	Run `dbt test --select +int_order_enriched+` (upstream + downstream tests)	Prevents silent breakage in other marts or alerts
6. Idempotency	The model is re-runnable (no side effects, safe to retry)	Ensure it uses `create or replace` or equivalent, not `insert into` without cleanup	Critical for recovery and orchestration reliability
7. Environment Parity	Dev/test environment reflects production data volume and stats	Use a production-like data subset or clone	Avoid being fooled by small dev datasets that run fast
8. Alerting & Monitoring	Freshness monitor and dbt test alerts are re-enabled post-fix	Confirm `fct_orders` freshness alert resets after successful run	Ensures future issues will be caught automatically
9. Documentation	Add a comment in the model or incident log explaining the fix	E.g., "Increased warehouse size; added join predicate to prevent blowup"	Prevents repeated firefighting; helps on-call

---

?? Example: If the Fix Was Query Optimization

Suppose we changed:
```sql
-- Before: Missing join condition
FROM orders o
JOIN customers c  -- Oops: no ON clause!
```
to:
```sql
-- After: Proper join
FROM orders o
JOIN customers c ON o.customer_id = c.id
```

Before shipping:
- ? Verify no cross-join in dev (check row count: should not jump from 1M -> 100B)
- ? Confirm `fct_orders` now runs and matches expected volume
- ? Check that Looker explores still serve data (via dev project)

? What NOT to Do
- ? Do not force a downstream backfill of `fct_orders` before `int_order_enriched` is stable
-

Engineering counterpart
6:57 AM - Responded in 0s

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

Agent response
6:58 AM - Responded in 23s

Excellent question. In any P1 production fix - especially under SLA pressure - understanding risks and having a rollback plan is essential to avoid turning a freshness incident into a data reliability crisis.

?? Top 5 Risks of Fixing `int_order_enriched`

Risk	Description	Potential Impact	Likelihood
1. Partial or Incorrect Data Fix	Optimization changes logic (e.g., wrong filter/join), causing data loss or duplication	`fct_orders` populates with bad data -> revenue misreported	? High
2. Cascading Failure	Fix introduces a syntax error or breaks dependent models (e.g., column dropped)	Multiple marts fail; blast radius expands	? Medium
3. Performance Improvements Don't Hold	Model runs fast in dev but fails again in prod due to data skew or concurrency	Timeout recurs -> SLA stays breached	? Medium
4. Orchestrator State Confusion	Manual runs or backfills interfere with scheduled DAG logic	Future runs skip, double-process, or hang	? Low-Medium
5. Downstream Caching Issues	Looker/Slack bot relies on stale cache; doesn't reflect new data even after fix	Perceived as "still broken" despite data being fixed	? Low

---

?? Rollback Plan (If Fix Fails)

We assume we deployed a change (e.g., optimized query), triggered `int_order_enriched`, but it:
- Still times out
- Runs but produces errors or bad data
- Breaks downstream models

? Immediate Rollback Steps

Step	Action	Goal
1	Revert the code change in Git and redeploy previous version of `int_order_enriched`	Return to known stable state
2	Manually unblock with last known good data (if safe) -> Query warehouse: does the last output of `int_order_enriched` still exist? Can it be reused?	Avoid total blackout
3	Force-refresh from last good upstream state -> Use `dbt run --vars {use_cached_int: true}` (if you have caching pattern)	Bypass broken model temporarily
4	Trigger downstream with manual override (only if data is acceptable)	Restore `fct_orders` while root cause is debugged
5	Pause dependent automations (e.g., Slack bot) to prevent noise	Reduce blast radius during recovery
6	Document and escalate to senior data engineer or on-call lead	Bring in deeper expertise

> ? Key Principle: Rollback != failure - it's a disciplined recovery. Better to revert and re-attempt than prolong bad data.

? Complementary Mitigations (Post-Rollback)

Once stable:
- ? Root cause analysis (RCA): Was this a code change? Data volume spike? Warehouse scaling issue?
- ? Implement guardrails:
  - Add query timeout alerts on long-running models

  - Enforce performance testing in CI (e.g., scan volume < 1TB for intermediate models)
  - Set max runtime thresholds in dbt + auto-kill
- ? Consider materializing `int_order_enriched` in smaller batches or partitioning to avoid full-table reprocessing

---

? When to Consider the Incident Resolved

Only after all of the following are true:
1. ? `int_order_enriched`

Engineering counterpart
6:58 AM - Responded in 0s

Thanks - upstream blocker and blast radius are well ordered. Prioritize revenue-facing dashboards if full freshness slips past the hour mark. Review complete - no further response needed.