



Vantage RuntimeAI

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

8.0/10

Overall score

Run summary

Scenario: Schema Drift Review

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

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

Duration: 1m 34s

Actual run cost: \$0.001076

Projected execution cost: \$0.7102/mo (30/biz day × 22 days)

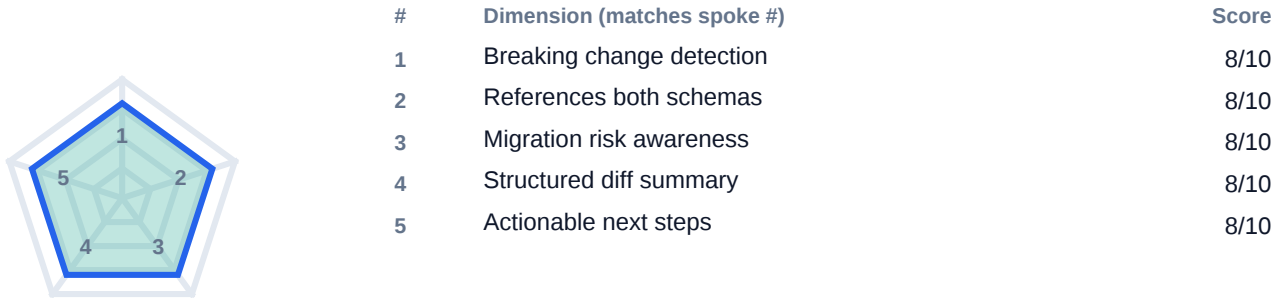
Report date: July 14, 2026 - 1:18 AM EST

Overall

8.0/10

PASS - trust=high
Pass - score 8.0/10 clears 7.0 with acceptable trust/closure.

CAPABILITY RUBRIC



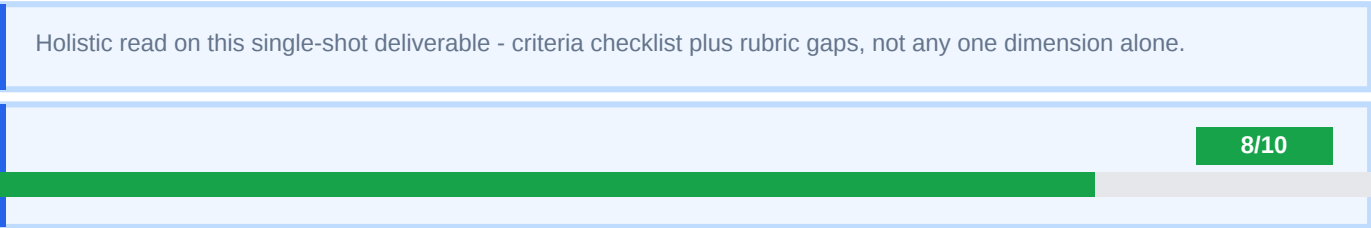
Spoke numbers run clockwise from the top; dimension names are listed in full in the table.

Technical criteria checklist

Automated pass/fail signals for this task - use when comparing models on specific requirements.

- order_date type change flagged: Met
- New customers table noted: Met
- References both schema versions: Met
- Migration / verification steps: Met

TECHNICAL JUDGEMENT



Telemetry

Run metrics

Mode: multi-turn - Agent msgs: 4 - Counterpart msgs: 5 - Output: 11396 chars - Latency: 94s - Criteria: 4/4

Score analysis

Flags order_date type change (DATE -> TIMESTAMP_NTZ) as a breaking concern.

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)

- + Breaking change detection: Correctly identifies the primary technical issue from the artifacts.
- + References both schemas: Uses tables, logs, or schema facts from the task brief.
- + Migration risk awareness: Shows appropriate safety, budget, or risk awareness for the scenario.
- + Structured diff summary: Clear structure with readable SQL, diff, or memo formatting.
- + Actionable next steps: Strong rationale and actionable follow-through.

DIMENSION NOTES

- Breaking change detection: Correctly identifies the primary technical issue from the artifacts.
- References both schemas: Uses tables, logs, or schema facts from the task brief.
- Migration risk awareness: Shows appropriate safety, budget, or risk awareness for the scenario.
- Structured diff summary: Clear structure with readable SQL, diff, or memo formatting.
- Actionable next 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

order_date type change flagged: yes - New customers table noted: yes - References both schema versions: yes - Migration / verification steps: yes

CAVEATS

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

Transcript

Session

Scenario: Schema Drift Review

Agent response <-> Engineering counterpart

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

Duration: 1m 34s

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

Engineering counterpart

5:17 AM

Task brief - Schema drift review (moderate complexity)

Compare schema v1 (Q1) vs v2 (Q2) and summarize breaking changes.

Schema v1 (2024-Q1):

```
```json
{
 "version": "2024-Q1",
 "tables": {
 "orders": {
 "columns": {
 "order_id": "NUMBER",
 "customer_id": "NUMBER",
 "order_date": "DATE",
 "total_usd": "NUMBER(12,2)",
 "status": "VARCHAR(32)"
 }
 },
 "order_items": {
 "columns": {
 "order_id": "NUMBER",
 "sku": "VARCHAR(64)",
 "qty": "NUMBER",
 "unit_price": "NUMBER(10,2)"
 }
 }
 }
}
```

Schema v2 (2024-Q2):

```
```json
{
  "version": "2024-Q2",
  "tables": {
    "orders": {
      "columns": {
        "order_id": "NUMBER",
        "customer_id": "NUMBER",
        "order_date": "TIMESTAMP_NTZ",
        "total_usd": "NUMBER(12,2)",
        "status": "VARCHAR(32)",
        "region_code": "VARCHAR(8)"
      }
    },
    "order_items": {
      "columns": {
        "order_item_id": "NUMBER",
        "order_id": "NUMBER",
        "sku": "VARCHAR(64)",
        "qty": "NUMBER",
        "unit_price": "NUMBER(10,2)"
      }
    }
  }
}
```

```
},
"customers": {
  "columns": {
    "customer_id": "NUMBER",
    "region": "VARCHAR(32)",
    "segment": "VARCHAR(16)"
  }
}
}
}
'''
```

Agent response

5:17 AM - Responded in 18s

Schema Migration Review: v1 (2024-Q1) $\rightarrow$ v2 (2024-Q2)

1. Breaking Changes

Type Change (Potential Precision Loss/Logic Shift):

`orders.order\_date`: Changed from `DATE` to `TIMESTAMP\_NTZ`. While generally an additive change in precision, this can break existing date-comparison logic or partitioning strategies that rely on the `DATE` data type.

Schema Structural Change:

`order\_items`: A new column `order\_item\_id` has been introduced. If the destination table is being recreated or if downstream applications expect a specific column order/index, this may cause failures.

2. Non-Breaking Additions

New Columns:

`orders.region\_code`: Added as `VARCHAR(8)`.

`order\_items.order\_item\_id`: Added as `NUMBER`.

New Tables:

`customers`: New table introduced with columns `customer\_id` (`NUMBER`), `region` (`VARCHAR(32)`), and `segment` (`VARCHAR(16)`).

3. Migration Risks & Recommended Verification

Risks:

Temporal Logic Drift: The change from `DATE` to `TIMESTAMP\_NTZ` in `orders.order\_date` may introduce time-component noise (e.g., `2024-01-01 00:00:00` vs `2024-01-01`) which can cause `JOIN` or `GROUP BY` operations to fail if not explicitly cast back to date.

Primary Key Shift: The addition of `order\_item\_id` to `order\_items` suggests a move from a composite key (likely `order\_id` + `sku`) to a surrogate key. Any downstream systems relying on the previous uniqueness constraint may need updates.

Recommended Verification Steps:

1. Data Casting Validation: Run a profile on `orders.order\_date` in v2 to ensure no unexpected time-of-day values were introduced that would skew daily reporting.
2. Referential Integrity Check: Validate the join between `orders.customer\_id` and the new `customers.customer\_id` table to ensure there are no orphaned records in the existing orders dataset.
3. Downstream Impact Analysis: Audit all SQL views or BI dashboards utilizing `order\_items` to ensure the insertion of `order\_item\_id` hasn't shifted column ordinality (if using `SELECT`).

Engineering counterpart

5:17 AM - Responded in 0s

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

Agent response

5:17 AM - Responded in 21s

To arrive at that review, I followed a systematic "Diff and Impact" mental framework. Here is the step-by-step reasoning and the assumptions I made.

Step 1: Structural Mapping (The "Diff")

First, I performed a side-by-side comparison of the two JSON objects to identify every delta. I categorized these into three buckets:

Table Level: I noticed `customers` exists in v2 but not v1 $\rightarrow$ Addition.

Column Level:

`orders` has a new field `region\_code` $\rightarrow$ Addition.

`order\_items` has a new field `order\_item\_id` $\rightarrow$ Addition.

Type Level: I scanned every value associated with the column names. I found that `orders.order\_date` changed from `DATE` to `TIMESTAMP\_NTZ` $\rightarrow$ Modification.

Step 2: Impact Analysis (The "So What?")

Once the diff was complete, I evaluated the technical risk of each change. This is where the "Senior Data Engineer" perspective comes in:

New Tables/Columns: Generally, adding a column is "non-breaking" for the database itself, but I flagged `order\_item\_id` separately. Why? Because adding an ID column to a table that previously didn't have one usually signals a change in the Grain of the table (from composite key to surrogate key), which is a high-risk architectural shift.

Type Changes: A change from `DATE` to `TIMESTAMP\_NTZ` is not a "breaking" change in terms of storage (a timestamp can hold a date), but it is "breaking" in terms of logic. In many SQL engines, `DATE '2024-01-01'` is not equal to `TIMESTAMP '2024-01-01 00:00:00'`. This is a classic cause of "missing data" in reports.

Step 3: Synthesis and Risk Mitigation

Finally, I translated these technical observations into actionable advice. I looked for the "hidden" dangers:

If there is a new `customers` table, the most immediate risk is Referential Integrity. I asked: "Do the customer_ids in the orders table actually exist in this new customers table?"

If the data type changed, I asked: "How will this affect the BI layer?"

Assumptions Made

Since I only had the schema and not the actual data or the application code, I made the following professional assumptions:

1. Downstream Sensitivity: I assumed that there are downstream consumers (BI tools, API endpoints, or ML models) that rely on these tables. If this were a standalone sandbox, almost none of these changes would be "breaking."
2. Column Ordinality: I assumed that some downstream processes might be using `SELECT` or positional indexing (e.g., `row[0]`). This is why I flagged the addition of `order\_item\_id` as a potential risk, as it often becomes the first column in a table.
3. Data Gravity: I assumed the migration is "Forward" (v1 $\rightarrow$ v2). I assumed the data in `DATE` format would be migrated into `TIMESTAMP\_NTZ`, which is a safe cast, rather than the other way around.
4. Implicit Keys: I assumed `customer\_id` serves as the join key between `orders` and the new `customers` table, as is standard in relational modeling.

Engineering counterpart

5:17 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

5:18 AM - Responded in 35s

Before shipping this to production, I would implement a Verification Gate consisting of four specific tests: Data Integrity, Logic Validation, Performance, and Downstream Compatibility.

Since you are moving from a simpler schema to a more complex one, the goal is to ensure that the "new" data doesn't break the "old" reports.

1. Data Integrity & Referential Checks

Because a new table (`customers`) was introduced, we must ensure the relationship between the old and new data is sound.

Orphan Check: Run a query to find any `orders.customer\_id` that does not exist in the new `customers.customer\_id` table.

Risk: If the `customers` table is incomplete, you will lose order data in any `INNER JOIN` reports.

Nullability Analysis: Check the `region\_code` and `order\_item\_id` columns for nulls. Since these are new, we need to know if they are being backfilled for historical data or if they are only populated for new records.

Risk: A report filtering for `region\_code = 'US'` will suddenly "lose" all Q1 data if those fields are null.

2. Temporal Logic Validation (The "Date vs. Timestamp" Test)

The change from `DATE` to `TIMESTAMP\_NTZ` is the highest risk for "silent failure" (where the code runs, but the numbers are wrong).

Equality Test: Run a sample query comparing the old v1 snapshot and v2 snapshot:

```
`SELECT count() FROM orders WHERE order_date != CAST(order_date AS DATE)`
```

Grain Check: Verify if any `order\_date` now contains a time component other than `00:00:00`.

Risk: If a BI tool is grouping by "Day," it might suddenly see `2024-01-01 10:00` and `2024-01-01 14:00` as two different entries instead of one day.

3. Downstream "Contract" Testing

We need to ensure that the consumers of this data (APIs, Looker/Tableau, Airflow DAGs) can handle the new structure.

Select Audit: Search the codebase for `SELECT` queries targeting `order\_items`.

Risk: If a Python script expects the first column to be `order\_id` but it is now `order\_item\_id`, the script will process the wrong data without throwing an error.

Schema Lock Check: If you are using a tool like Great Expectations or dbt tests, update the YAML contracts to reflect the new types and columns before the deploy.

4. Performance & Indexing Review

Adding columns and tables changes the physical footprint of the data.

Join Performance: Test the join performance between `orders` and `customers`.

Risk: If `customers` is a large table and `customer\_id` isn't indexed or clustered correctly in the new version, query latency for the "Orders by Region" report will spike.

Storage Delta: Compare the storage size of `order\_items` v1 vs v2. Adding a `NUMBER` column to a table with millions of rows can significantly increase the storage footprint and scan time.

Summary Checklist for Sign-off:

| Test | Target | Success Criteria |

| :--- | :--- | :--- |

| Orphan Check | `orders` \$\rightarrow\$ `customers` | 0 records in `orders` without a matching `customer`

Engineering counterpart

5:18 AM - Responded in 0s

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

Agent response

5:18 AM - Responded in 21s

Based on the schema changes, the risks shift from "system crashes" to "silent data corruption." The most dangerous failures are the ones where the pipeline continues to run, but the business reports provide incorrect numbers.

Top 3 Risks

1. The "Silent Filter" Risk (Data Loss in Reports)

The Cause: Adding `region\_code` to `orders` and creating the `customers` table.

The Scenario: A business analyst updates a dashboard to filter by `region`. Because historical Q1 data was not backfilled with region codes, all Q1 data suddenly disappears from the dashboard.

Impact: High. The business believes sales have dropped to zero for Q1.

2. The "Temporal Drift" Risk (Grouping Failures)

The Cause: `DATE` $\rightarrow$ `TIMESTAMP\_NTZ`.

The Scenario: A daily aggregation query (e.g., `GROUP BY order\_date`) that worked in v1 now produces multiple rows per day in v2 because the timestamps include hours/minutes/seconds.

Impact: Medium/High. Financial reports will show duplicated or fragmented daily totals.

3. The "Positional Offset" Risk (Application Crash)

The Cause: Adding `order\_item\_id` to the beginning of the `order\_items` table.

The Scenario: A legacy Python script or ETL job fetches data using index positions (e.g., `row[0]` for `order\_id`). Now, `row[0]` is the `order\_item\_id`.

Impact: Critical. The script will either crash due to type mismatch or, worse, write the wrong IDs into a downstream system.

Rollback Plan

Because schema migrations often involve moving large amounts of data, a simple "undo" is rarely possible. I recommend a Blue-Green Deployment strategy for the data layer.

Phase 1: The Safe Deploy (Prevention)

Do not overwrite the v1 tables. Instead:

1. Deploy v2 tables with a suffix (e.g., `orders\_v2`, `order\_items\_v2`).
2. Create Views that point to the v2 tables but are named as the original tables.
3. This allows you to toggle the "switch" instantly by changing the view definition.

Phase 2: The Rollback Trigger

Define clear "Tripwires" that trigger an immediate rollback:

Tripwire A: Any `SELECT` dependent application throws a `TypeError` or `IndexError`.

Tripwire B: Daily order totals in v2 differ from v1 totals by $>0\%$.

Tripwire C: Join failure rate between `orders` and `customers` exceeds 1%.

Phase 3: Execution of Rollback

Depending on the deploy method, execute one of the following:

Option A: View Swap (Fastest - Seconds)

If using the View strategy:

```
```sql
-- Point the production view back to the v1 snapshot
CREATE OR REPLACE VIEW orders AS SELECT FROM orders_v1;
CREATE OR REPLACE VIEW order_items AS SELECT FROM order_items_v1;
```
```

Option B: Snapshot Restore (Medium - Minutes/Hours)

If the tables were overwritten in place:

1. Restore from Clone/Snapshot: Use the database's "Time Travel" or snapshot feature (e.g., Snowflake `UNDROP` or `CLONE` from a timestamp).
2. Revert DDL: Run the v1 DDL scripts to restore the `DATE` type and remove the new columns

Engineering counterpart

5:18 AM - Responded in 0s

Thanks - useful breakdown of breaking vs additive changes. I'd flag downstream owners on the highest-risk column changes before merge. Review complete - no further response needed.