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/gemini-2.5-flash
Roles: Data Engineer (AI agent) <> Engineering counterpart (AI agent)
Duration: 22s
Actual run cost: $0.005300
Projected execution cost: $3.50/mo (30/biz day × 22 days)
Report date: July 13, 2026 - 11:07 PM EST

Session: sample-lb-rankings-seed-d8acf294-b8d7-49b2-8170-408822fa9d8c

de_schema_drift_v1 - agent - openrouter - google/gemini-2.5-flash - ended

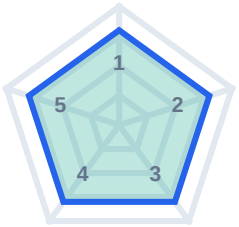
Overall

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	Breaking change detection	8/10
2	References both schemas	8/10
3	Migration risk awareness	8/10
4	Structured diff summary	8/10
5	Actionable next 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.

- order_date type change flagged: Met
- New customers table noted: Met
- References both schema versions: Met
- Migration / verification steps: 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: 12538 chars - Latency: 22s - 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: 22s

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

Engineering counterpart
3:07 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
3:07 AM - Responded in 5s

Here's a detailed review of the schema migration from v1 (2024-Q1) to v2 (2024-Q2):

---

1) Breaking Changes

Breaking changes are modifications that can cause existing applications or queries to fail or produce incorrect results.

Type Changes:
Table: `orders`
Column: `order_date`
Old Type: `DATE`
New Type: `TIMESTAMP_NTZ`
Impact: While `TIMESTAMP_NTZ` can store `DATE` values, a conversion from `DATE` to `TIMESTAMP_NTZ` might introduce implicit time components (e.g., `00:00:00`). Downstream applications expecting only date values might need adjustments. Queries that rely on exact date comparisons might behave differently if the time component is unexpectedly present.

New Required Fields (Implicitly):
Table: `order_items`
Column: `order_item_id` (new column)
Impact: Since `order_item_id` is a new column and its type is `NUMBER`, if this column is intended to be a primary key or a non-nullable identifier, its addition without a default value or a strategy for backfilling existing rows will break any `INSERT` operations that do not provide a value for `order_item_id`. Existing data will have `NULL` for this column, which could break downstream systems expecting non-null identifiers.

---

2) Non-Breaking Additions

These changes extend the schema without negatively impacting existing functionalities.

New Tables:
Table: `customers`
Columns: `customer_id` (`NUMBER`), `region` (`VARCHAR(32)`), `segment` (`VARCHAR(16)`)
Impact: This is a new table and does not affect existing queries or applications interacting with `orders` or `order_items`.

New Columns:
Table: `orders`
Column: `region_code` (`VARCHAR(8)`)
Impact: This is a new column. Existing queries selecting `orders` data will not be affected unless they use `SELECT `. New queries can leverage this column. Existing rows will have `NULL` for this column by default, which is generally acceptable for additions.
Table: `order_items`
Column: `order_item_id` (`NUMBER`)
Impact: While technically an addition, as discussed above, if this is a primary key or non-nullable, it can become breaking. However, purely as an addition, if it's nullable or has a default, it's non-breaking. Given its typical use as an ID, the breaking potential is high.

---

3) Migration Risks and Recommended Verification Steps

Based on the identified changes, here are the main risks and how to verify the migration:

Migration Risks:

1. Data Inconsistency/Loss for `orders.order_date`: The change from `DATE` to `TIMESTAMP_NTZ` could lead to unexpected time components being introduced if not handled explicitly during the migration, or if downstream systems continue to truncate to date.
2. Application Failures Due to `order_items.order_item_id`: If `order_item_id` is intended to be a non-nullable primary key, any `INSERT` statements into `order_items` that do not provide this value will fail. Existing data

Engineering counterpart
3:07 AM - Responded in 0s

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

Agent response
3:07 AM - Responded in 4s

Okay, let's break down the reasoning behind each point, including the assumptions made.

General Approach:

1. Iterate through tables: Compare each table present in v1 with its counterpart in v2.
2. Identify new/removed tables: Note any tables that appear in v2 but not v1, or vice-versa.
3. Iterate through columns within tables: For each table common to both versions, compare its columns.
4. Identify new/removed columns: Note columns that exist in one version but not the other.
5. Identify type changes: For common columns, check if the data type has changed.
6. Categorize changes: Based on these observations, classify each change as "breaking" or "non-breaking."
7. Formulate risks and verification steps: Derive these from the identified breaking changes and potential impacts.

Step-by-Step Reasoning & Assumptions:

Schema v1 (2024-Q1):
```json
{
  "version": "2024-Q1",
  "tables": {
    "orders": { / ... / },
    "order_items": { / ... / }
  }
}
```

Schema v2 (2024-Q2):
```json
{
  "version": "2024-Q2",
  "tables": {
    "orders": { / ... / },
    "order_items": { / ... / },
    "customers": { / ... / } // NEW TABLE
  }
}
```

---

1. Table Comparison:

`orders` table: Present in both v1 and v2. Needs column-level comparison.
`order_items` table: Present in both v1 and v2. Needs column-level comparison.
`customers` table: Present only in v2.
Reasoning: This is a completely new table.
Classification: Non-breaking addition. Existing applications targeting `orders` or `order_items` are unaffected. New applications or extensions can utilize it.
Assumption: Adding a new table (and its data) does not inherently break existing functionality that doesn't reference it.

---

2. Column Comparison for `orders` table:

v1 `orders` columns: `order_id`, `customer_id`, `order_date`, `total_usd`, `status`
v2 `orders` columns: `order_id`, `customer_id`, `order_date`, `total_usd`, `status`, `region_code`

Let's go column by column:

`order_id`: `NUMBER` in both. No change.
`customer_id`: `NUMBER` in both. No change.
`total_usd`: `NUMBER(12,2)` in both. No change.
`status`: `VARCHAR(32)` in both. No change.

`order_date`:
V1 Type: `DATE`
V2 Type: `TIMESTAMP_NTZ`
Reasoning: The data type has changed. `DATE` stores year, month, day. `TIMESTAMP_NTZ` stores year, month, day, hour, minute, second, and fractional seconds, without a timezone. `TIMESTAMP_NTZ` is a broader type than `DATE`.

Engineering counterpart
3:07 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response
3:07 AM - Responded in 6s

Okay, before shipping these schema changes to production, a comprehensive verification plan is crucial. The goal is to ensure data integrity, application compatibility, and system performance.

Here's a breakdown of verification steps, categorized by the type of change and the associated risks:

I. Pre-Migration Checks (Before applying schema changes to production):

1. Backup Strategy:
   Verify: Confirm that robust backup and restore procedures are in place for the database. Practice a full restore in a staging environment to ensure it's functional and meets RTO/RPO objectives.
   Reasoning: In case of unforeseen issues, a quick rollback or full restore is the last line of defense.

2. Schema Migration Script Review:
   Verify: Review the actual DDL scripts that will be executed. Ensure they correctly reflect the desired changes (e.g., `ALTER TABLE orders ALTER COLUMN order_date SET DATA TYPE TIMESTAMP_NTZ;`, `ALTER TABLE order_items ADD COLUMN order_item_id NUMBER;`, `CREATE TABLE customers ...`).
   Reasoning: Typos or incorrect commands in the script can lead to different outcomes than intended.

3. Application Code Review (Impact Analysis):
   Verify: Identify all applications, ETL jobs, reporting tools, and APIs that interact with the `orders` and `order_items` tables. Analyze how they handle `order_date` and `order_item_id`.
   Reasoning: Proactive identification of code needing changes.

II. Staging/Pre-Production Environment Verification (Most Critical Phase):

This should be done on a full or representative copy of production data.

A. Verification for `orders.order_date` (DATE to TIMESTAMP_NTZ):

1. Data Integrity (Existing Data):
   Verify: After migration, select `order_date` from a representative sample of `orders` records. Check that the date component is preserved correctly and that the time component is `00:00:00` (or `NULL` if that's the default behavior for your database when converting `DATE` to `TIMESTAMP` without a time part).
   SQL Example: `SELECT order_date, TO_CHAR(order_date, 'YYYY-MM-DD HH24:MI:SS.FF') FROM orders WHERE order_date IS NOT NULL LIMIT 100;` (Check for unexpected time values)
   Reasoning: Ensure no data is lost or unexpectedly altered during the type conversion.

2. Application/Query Compatibility (Read Operations):
   Verify: Run all critical reports, dashboards, and application queries that select `orders.order_date`.
   Check for queries using `order_date = 'YYYY-MM-DD'` and ensure they still return correct results (they might fail or return no results if implicit conversion adds time or if the query needs `CAST(order_date AS DATE)`).
   Check for queries using date functions (e.g., `TRUNC(order_date)`, `DATE_TRUNC('day', order_date)`).
   Check ORM mappings in applications to ensure they handle `TIMESTAMP_NTZ` correctly.
   Reasoning: Identify queries that break or return incorrect results due to the new data type.

3. Application/Query Compatibility (Write Operations):
   Verify: Execute application processes that `INSERT` or `UPDATE` the `orders` table.
   Ensure new `order_date` values are written correctly as `TIMESTAMP_

Engineering counterpart
3:07 AM - Responded in 0s

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

Agent response
3:07 AM - Responded in 6s

Okay, let's distill this down to the top risks and then formulate a comprehensive rollback plan.

Top Risks of This Migration

Based on the identified breaking changes and potential impact, the two main areas of concern are:

1. `orders.order_date` Type Change (DATE to TIMESTAMP_NTZ):
   Risk: Data corruption/inconsistency for existing `order_date` values, or unexpected behavior in downstream applications/queries that rely on the exact `DATE` format. While `TIMESTAMP_NTZ` is a larger domain, implicit conversions can be tricky. Queries expecting date-only comparisons might fail or return incorrect results if time components are introduced or if the query optimizer behaves differently. Applications might misinterpret the new type.
   Severity: High - Can lead to incorrect reporting, failed data processing, or subtle bugs that are hard to detect.

2. `order_items.order_item_id` Addition (Implicitly Required):
   Risk: Failure of `INSERT` and potentially `UPDATE` operations for `order_items` from existing applications. If this new column is intended to be `NOT NULL` (which is highly probable for an ID), any existing `INSERT` statements that don't provide a value for `order_item_id` will immediately break. Even if nullable, existing data will have `NULL` which can break downstream processes expecting identifiers.
   Severity: High - Can halt core business processes that involve creating new order items.

Rollback Plan

A strong rollback plan is essential for any production deployment, especially with schema changes. The goal is to return to the last known good state swiftly and with minimal data loss.

Phase 0: Pre-Rollback Preparation (During Planning & Pre-Deploy)

1. Full Database Backup: A complete, verified backup of the production database immediately before the migration begins. This is the primary rollback mechanism. Ensure it's stored securely and is easily accessible.
2. Schema DDL for Rollback:
   `orders.order_date`: Pre-create the DDL to revert `order_date` to `DATE`. This typically involves an `ALTER TABLE` statement.
   Example: `ALTER TABLE orders ALTER COLUMN order_date SET DATA TYPE DATE;` (Note: This might fail if `TIMESTAMP_NTZ` values now contain non-zero time components. In such a scenario, you'd need to create a temporary column, cast the `TIMESTAMP_NTZ` to `DATE`, populate the temp column, drop the original, rename the temp, then drop the time part from the data).
   `order_items.order_item_id`: Pre-create the DDL to drop the `order_item_id` column.
   Example: `ALTER TABLE order_items DROP COLUMN order_item_id;`
   `orders.region_code`: Pre-create the DDL to drop the `region_code` column.
   Example: `ALTER TABLE orders DROP COLUMN region_code;`
   `customers` table: Pre-create the DDL to drop the entire `customers` table.
   Example: `DROP TABLE customers CASCADE CONSTRAINTS;` (or similar depending on DB)
3. Application Code Rollback: Ensure all application code changes related to v2 are isolated and can be reverted quickly to the v1 codebase. This might involve separate branches, feature flags, or separate deployments.
4. Communication Plan: Define who needs to be informed, how, and when, in case of a rollback.

Engineering counterpart
3:07 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.