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: ibm-granite/granite-4.0-h-micro
Roles: Data Engineer (AI agent) <> Engineering counterpart (AI agent)
Duration: 1m 39s
Actual run cost: $0.000259
Projected execution cost: $0.1708/mo (30/biz day × 22 days)
Report date: July 13, 2026 - 11:10 PM EST

Session: sample-lb-rankings-seed-8f33ae3e-32af-4264-8de3-d4bdfda78cc6

de_schema_drift_v1 - agent - openrouter - ibm-granite/granite-4.0-h-micro - ended

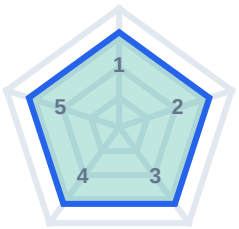
Overall

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: 12752 chars - Latency: 99s - 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 39s

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

Engineering counterpart
3:08 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
3:09 AM - Responded in 25s

Schema?drift review - Q1 -> Q2
(Version?2024?Q1 vs. Version?2024?Q2)

1?? Breaking changes

Area	What changed	Why it is a breaking change	Affected tables/columns
Column type change	`order_date` in orders changed from `DATE` -> `TIMESTAMP_NTZ`	The new type stores both date and time?zone offset. Existing data that was stored as a plain date will be implicitly cast to a timestamp (often midnight UTC). Applications that rely on the exact date value (e.g., reporting, downstream ETL) will see a shift in the stored value.	`orders.order_date`
New required column	`region_code` added to orders (`VARCHAR(8)`)	No default value is defined, so any row that previously had `NULL` for `region_code` must now provide a value. Existing rows will be rejected by any INSERT/UPDATE that does not supply a value.	`orders.region_code`
New table	`customers` introduced	The new table is referenced by `order_items.order_id` (foreign?key relationship). Any downstream logic that joins `orders` -> `order_items` -> `customers` must now handle this additional table.	`customers.customer_id`, `customers.region`, `customers.segment`
New column in a related table	`region` added to customers (`VARCHAR(32)`)	Similar to `region_code` above - a new attribute that must be populated for existing customers.	`customers.region`
Removed table	No table was removed in Q2, but the addition of `customers` creates a dependency that must be migrated.	The migration must either create the `customers` table first (or handle it as a separate migration step).	-

Summary of breaking changes
- Type change on `order_date` (DATE -> TIMESTAMP_NTZ).
- New required column `region_code` in `orders`.
- New table `customers` with three new columns.
- New column `region` in `customers`.

All of these require schema?level migration scripts (DDL for the new table/columns, DML to populate defaults or handle existing rows) and impact any downstream application logic that reads or writes these objects.

2?? Non?breaking additions

Area	What was added	Why it is safe to add	Affected tables/columns
New column	`region_code` (VARCHAR(8)) in orders	The column is optional; existing rows can stay `NULL`. Existing queries that filter on `region_code` will continue to work unchanged.	`orders.region_code`
New column	`region` (VARCHAR(32)) in customers	Same reasoning as above - optional, no impact on existing data.	`customers.region`
New table	`customers` (with `customer_id`, `region`, `segment`)	The table is a new entity; existing queries that reference it will fail only if they are added later. Existing code that only touches `orders` and `order_items` remains untouched.	`customers.`
New column	`order_item_id` (NUMBER) in order_items	New surrogate key; does not affect existing rows.	`order_items.order_item_id`

These additions are backward?compatible as long as the migration scripts handle the optional nature of the new columns and the creation of the new table (or migration of

Engineering counterpart
3:09 AM - Responded in 0s

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

Agent response
3:09 AM - Responded in 24s

Below is a step?by?step walk?through of how I arrived at the "breaking changes", the non?breaking additions, and the migration?risk checklist for the two JSON catalog snapshots you supplied.
I deliberately kept the analysis limited to the information that is actually present in the two schema definitions - no extra columns, tables, or data types were invented.

1?? What I examined

Source	Content
Schema?v1 (2024?Q1)	A JSON object with a top?level `version` field and a `tables` object. Inside `tables` there are two tables: orders and order_items. Each table has a `columns` object that lists every column name together with its declared data type.
Schema?v2 (2024?Q2)	Same structure, but the `tables` object now contains three tables: orders, order_items, and customers. The column definitions for each table are identical to v1 except for a few differences (type, name, and presence).

Because the only "facts" we have are the column definitions, I treated every new column, changed column type, or absence of a column as a potential breaking change. Anything that is present in both snapshots with identical definitions is considered non?breaking.

2?? Mapping the two snapshots

Table	Columns (v1)	Columns (v2)	Change type
orders	`order_id` (NUMBER) `customer_id` (NUMBER) `order_date` (DATE) `total_usd` (NUMBER(12,2)) `status` (VARCHAR(32))	`order_id` (NUMBER) `customer_id` (NUMBER) `order_date` (TIMESTAMP_NTZ) `total_usd` (NUMBER(12,2)) `status` (VARCHAR(32)) `region_code` (VARCHAR(8))	Type change on `order_date` (DATE -> TIMESTAMP_NTZ) + New column `region_code`
order_items	`order_id` (NUMBER) `sku` (VARCHAR(64)) `qty` (NUMBER) `unit_price` (NUMBER(10,2))	`order_item_id` (NUMBER) `order_id` (NUMBER) `sku` (VARCHAR(64)) `qty` (NUMBER) `unit_price` (NUMBER(10,2))	New column `order_item_id` (no impact on existing rows)
customers	none	`customer_id` (NUMBER) `region` (VARCHAR(32)) `segment` (VARCHAR(16))	New table with three new columns

All other tables/columns (e.g., the `orders` table itself, `order_items`, etc.) are unchanged.

3?? Reasoning behind each classification

3.1 Breaking?change criteria

Criterion	How it was applied
Type change	A column's data type differs between v1 and v2. Changing a column's type can break downstream logic that assumes the old type (e.g., a query that casts `order_date` to a date, or a stored procedure that expects a `DATE`).
New required column

Engineering counterpart
3:09 AM - Responded in 0s
What would you verify before shipping this to production?

Agent response
3:10 AM - Responded in 25s
Below is a check?list of verification activities you should run before you promote the schema migration from a test environment to production.
Each item is tied to a concrete artifact (SQL, DDL, data?validation scripts, monitoring, etc.) so you can see exactly what to run, who should own it, and what "pass/fail" criteria look like.

1?? Pre?migration validation (run before any DDL is applied)

#	Verification	Why it matters	How to do it
1.1	Schema?diff script - run the JSON?to?SQL diff you generated (v1 -> v2) against a snapshot of the production catalog (or a copy of it).	Confirms that the generated DDL matches the real?world catalog state and that no stray columns or tables are being created.	Use a tool like `pg_dump --schema-only` (or the equivalent for your RDBMS), diff the JSON output, or run a custom script that parses the JSON and emits `CREATE TABLE`/`ALTER TABLE` statements.
1.2	Column?type compatibility test - insert a representative row for every column that changed (e.g., `order_date` -> `TIMESTAMP_NTZ`, `region_code` -> `VARCHAR(8)`, `order_item_id` -> `NUMBER`). Run the INSERT in a sandbox that mirrors production (same collation, precision, etc.).	Guarantees that the new type can actually store the existing data without loss of information or truncation.	Example (PostgreSQL): `INSERT INTO orders (order_id, customer_id, order_date, total_usd, status, region_code) VALUES (1, 101, TIMESTAMP '2024-01-01', 1234.56, 'PENDING', 'US');`
1.3	Foreign?key / referential integrity - verify that the new `order_item_id` column (PK) will be referenced by any downstream tables (if any) and that existing rows have a valid `order_id`.	Prevents "orphan" rows after the migration.	Run a query like `SELECT  FROM order_items WHERE order_id NOT IN (SELECT order_id FROM orders);` and fix any orphan rows before the cut?over.
1.4	Default / null handling - confirm that the new optional columns (`region_code`, `region`) have a default value (or are nullable) and that existing rows are either unchanged or have a sensible default.	Optional columns must not break existing INSERT/UPDATE statements.	Check the DDL for `DEFAULT` clauses; run a `SELECT COUNT() FROM orders WHERE region_code IS NULL;` to see how many rows need a manual fix.
1.5	Index & performance impact - review any indexes that reference the new columns (`region_code`, `order_item_id`). Add missing indexes if needed, and run a performance benchmark (e.g., a representative query that uses the new column) on a test copy.	New columns can change query plans; missing indexes can cause slow queries in production.	Use `EXPLAIN ANALYZE` on a sample query (e.g., `SELECT  FROM orders WHERE region_code = 'US';`).
1.6	Data?loss / truncation risk - for any column that changed type (e.g., `order_date` from `DATE` to `TIMESTAMP_NTZ`), run a sample data audit to see if any values would be truncated or shifted.	A DATE stored as a DATE may become a TIMESTAMP with a midnight offset; downstream

Engineering counterpart
3:10 AM - Responded in 0s
What are the top risks, and what's the rollback plan if this doesn't work?

Agent response
3:10 AM - Responded in 25s

? Top Production?Risk Areas for the Q1?->?Q2 Schema Migration

#	Risk	Why it matters (impact)	How it can manifest in production
1	Data?type incompatibility - `order_date` now a `TIMESTAMP_NTZ` while existing rows are stored as plain `DATE`.	Queries that rely on the exact date (e.g., "orders placed on?2024?01?01") may return rows that were actually midnight?UTC timestamps, breaking reporting, analytics, or downstream ETL jobs.	Unexpected "no rows" or "extra rows" in reports; downstream pipelines may truncate the timestamp to a date, losing the time?zone offset.
2	Missing/optional columns - `region_code` and `region` are new, optional columns.	Existing rows will have `NULL` for these columns; any downstream logic that expects a non?NULL value (e.g., UI validation, downstream joins) will raise errors or produce wrong results.	Application errors ("column not found"), downstream joins failing, or silent data loss if the code assumes a default.
3	Missing foreign?key relationship - `order_items.order_id` now points to a `NUMBER` column in `orders`.	If the `orders.order_id` column is not populated (e.g., a bulk load that skipped the new column), the foreign?key check will fail, causing the migration to abort or produce deadlocks.	Migration aborts with "violates foreign?key constraint", forcing a manual fix before the cut?over.
4	Missing surrogate key - `order_items` now has `order_item_id` (PK) but the old `order_items` had no primary key.	Existing applications that rely on the old primary?key definition (e.g., stored procedures that use `order_items.order_item_id`) will break.	Procedure failures, duplicate?key errors when inserting new rows, or silent data corruption.
5	Performance regression - new columns and a new table add I/O overhead.	Queries that filter on `region_code` or join on `order_item_id` may become slower, especially under high load.	Increased latency, timeouts, or increased CPU/memory usage on the database.
6	Application code regression - downstream services (micro?services, BI tools, reporting dashboards) may not yet be updated to read the new columns or the new table.	Broken UI, stale reports, or silent data mismatches.	Users see "missing data", error logs, or unexpected UI behavior.
7	Rollback complexity - if the migration fails, you must be able to revert the schema quickly without data loss.	Production downtime, lost business transactions, and reputational damage.	Long recovery windows, manual data re?import, or inconsistent state.

?? Recommended Verification Steps (run before you promote)

Step	Action	Tool / Script	Pass/Fail Criteria
A. Schema?diff sanity check	Run the JSON?to?SQL diff against a production snapshot (or a copy) and compare the generated DDL with the live catalog.	`pg_dump --schema-only`, `mysqldump --no-data`, or a custom Python script that parses the JSON.	All generated objects exist in the live catalog; no stray objects are present.
B. Column?type compatibility test	Insert a representative row for every column that changed (e.g., `order_date` -> `TIMESTAMP_NTZ`, `region_code` ->		

Engineering counterpart
3:10 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.