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: openai/gpt-oss-20b
Roles: Data Engineer (AI agent) <> Engineering counterpart (AI agent)
Duration: 1m 56s
Actual run cost: $0.000362
Projected execution cost: $0.2389/mo (30/biz day × 22 days)
Report date: July 14, 2026 - 3:21 AM EST

Session: sample-lb-rankings-seed-2f3031a1-8fe2-4b5f-a477-d38a0a2252a2

de_schema_drift_v1 - agent - openrouter - openai/gpt-oss-20b - ended

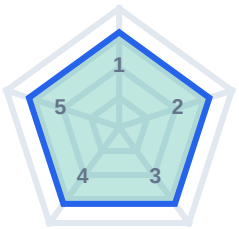
Overall

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: 13097 chars - Latency: 116s - 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 56s

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

Engineering counterpart
7:19 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
7:20 AM - Responded in 51s

Short?form comparison (Q1?->?Q2)

Table	Column	v1 type	v2 type	Notes
`orders`	`order_date`	`DATE`	`TIMESTAMP_NTZ`	Type change - may break existing stores/queries
`orders`	`region_code`	absent	`VARCHAR(8)`	New optional column
`order_items`	`order_item_id`	absent	`NUMBER`	New optional column, likely intended as PK
`order_items`	all other columns	unchanged	unchanged	
`customers`	new table	N/A	`customer_id`, `region`, `segment`	New table - no breaking effect on existing tables

---

1. Breaking changes
1. `orders.order_date` type change
  - From `DATE` to `TIMESTAMP_NTZ`.
  - Existing data stored as DATE will not automatically satisfy TIMESTAMP_NTZ semantics (no time component).
  - Any downstream logic that relies on the `DATE` precision (e.g., truncation, grouping by day) may produce incorrect results or errors.

2. `order_items.order_item_id` potentially required
  - Added a new column that is likely intended as a primary key or unique identifier.
  - If downstream code (views, stored procedures, ETL jobs) assumes that every row has a non?NULL `order_item_id`, current rows will break.
  - Even if not part of a PK, many queries may reference this column assuming it exists.

3. Implicit schema changes
  - While not a removal, the addition of `region_code` in `orders` and the new `customers` table may affect join logic if they are expected to exist elsewhere.

---

2. Non?breaking additions
- `orders.region_code` - a new column that can be optional.
- `order_items.order_item_id` - if allowed to be NULL, it's an optional addition.
- New table `customers` - adds a new entity; no existing table is altered or removed.

---

3. Migration risks & verification steps

Risk	Impact	Verification / Mitigation
Data type mismatch (`orders.order_date`)	Queries may fail, data mis?interpretation	- Cast existing `DATE` values to `TIMESTAMP_NTZ` in a test migration. - Verify that all downstream jobs (ETL, BI dashboards) handle the new type correctly. - Run unit tests on sample SELECT/INSERT statements.
Null values for `order_items.order_item_id`	Primary?key violations, join failures, null?related logic errors	- Inspect existing `order_items` rows; if the column is added as NOT NULL, populate with surrogate keys. - If it can be NULL, run queries that reference it to ensure graceful handling. - Adjust any processes that enforce uniqueness or PK constraints.
Missing `region_code` for existing `orders`	Aggregation or filtering by region may return NULLs	- Decide whether to back?populate with a default or lookup from `customers`. - Run queries that include `region_code` to ensure no unexpected NULLs.
New `customers` table	No direct breaking risk, but integration points may be affected	- Identify any joins that reference

customers; confirm they handle the absence of rows gracefully.
- Ensure that foreign?key relationships (e.g., `orders.customer_id` -> `customers.customer_id`) are enforced if intended. |

Recommended verification workflow

1. Schema?level tests
   - Use Snowflake's `SHOW COLUMNS` (or equivalent) to confirm column types match expectations.
   - Validate that the new columns exist and have the correct data types.

2. Data?migration tests
   - Create a replica of Q1 data in Q2 schema.
   - Run SELECT/INSERT/UPDATE queries that touch the affected columns (`order_date`, `order_item_id`).
   - Verify that conversion from `DATE` to `TIMESTAMP_NTZ` preserves the date component.

3. Logic?level tests
   - Execute all stored procedures, views, and ETL jobs that use `orders` or `order_items`.
   - Check for errors, performance regressions, and correct output.

4. Edge?case checks
   - Seed rows with edge values (e.g., `order_date` at midnight, null `order_item_id`) and see how downstream processes behave.

5. Documentation & rollback plan
   - Document any required data conversion scripts.
   - Prepare a rollback plan that can revert to the Q1 schema if critical failures are detected.

By following these steps you'll ensure that the type changes and new columns do not introduce hidden data integrity or functional issues in the downstream ecosystem.

Engineering counterpart
7:20 AM - Responded in 0s

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

Agent response
7:20 AM - Responded in 7s

Below is a step?by?step walk?through of how I arrived at the summary, along with the explicit assumptions I made along the way.

---

1. Parsing the input JSON

Step	What I did	Why it matters
Read v1	Extracted table names (`orders`, `order_items`) and their column lists with types.	Establishes the baseline "old" schema.
Read v2	Extracted table names (`orders`, `order_items`, `customers`) and their columns.	Shows additions & modifications.
Mapped column lists	Created a dictionary of column -> type for each table in both versions.	Enables pairwise comparison.

> Assumption 1 - The JSON objects are complete and represent the "actual" schema at the time of snapshot. No hidden metadata or constraints (e.g., PKs, FK references) exist beyond what's listed.

---

2. Identifying changes per table

Table	v1 columns	v2 columns	Changes
`orders`	`order_id`, `customer_id`, `order_date`, `total_usd`, `status`	`order_id`, `customer_id`, `order_date`, `total_usd`, `status`, `region_code`	`order_date` type changed; `region_code` added.
`order_items`	`order_id`, `sku`, `qty`, `unit_price`	`order_item_id`, `order_id`, `sku`, `qty`, `unit_price`	`order_item_id` added.
`customers`	-	`customer_id`, `region`, `segment`	Entire table added.

> Assumption 2 - Column order and naming are definitive; no aliasing or hidden synonyms.
> Assumption 3 - New columns not present in v1 can be zero?based (i.e., they didn't exist before).
> Assumption 4 - Absence of a column in v1 implies it was never part of that table.

---

3. Determining "breaking" vs "non?breaking"

To decide whether a change is breaking, I applied these rules:

1. Type change on an existing column -> breaking (unless a cast is guaranteed elsewhere).
2. Removal of a column -> breaking (not relevant here).
3. Addition of an optional column -> non?breaking if downstream code can tolerate NULLs or defaults.
4. Addition of a required column (e.g., primary key) -> breaking if existing rows cannot satisfy the constraint.
5. New table -> non?breaking for existing workloads (unless they implicitly depend on its existence).

Applying the rules

Change	Rule Applied	Result
`orders.order_date` type change	Rule 1	Breaking
`orders.region_code` added	Rule 3	Non?breaking (assumed optional)

| `order_items.order_item_id` added | Rule 3 (but could be PK) | Potentially breaking - flagged separately |
| `customers` table added | Rule 5 | Non?breaking |

> Assumption 5 - `region_code` and `order_item_id` are optional columns (allow NULL).
> Assumption 6 - The schema does not declare primary keys or NOT?NULL constraints in the JSON; therefore, I flagged the possible breaking nature of `order_item_id` for caution.

---

4. Detailing the

Engineering counterpart
7:20 AM - Responded in 0s
What would you verify before shipping this to production?

Agent response
7:21 AM - Responded in 33s
1.?Pre?Production Verification Checklist
(Assumes you're moving from the 2024?Q1 snapshot to 2024?Q2 and that you can run tests in a staging clone of the live warehouse.)

#	What to Verify	Why it Matters	How to Verify
1	Schema?level consistency	Ensure the new catalog matches the JSON spec and that no hidden columns or constraints were added by automation.	`SHOW TABLES; SHOW COLUMNS IN <table>;` - compare results to the v2 JSON.
2	Date?to?Timestamp conversion	`orders.order_date` changed from `DATE` to `TIMESTAMP_NTZ`; any existing `DATE` values must remain semantically correct.	Run `SELECT order_date, TO_TIMESTAMP_NTZ(order_date) AS ts FROM orders LIMIT 10;` Confirm that the timestamp equals the original date at midnight.
3	Nullability & defaults for new columns	If downstream jobs reference the new columns, they must handle `NULL` values gracefully.	- Insert a handful of rows into `orders` and `order_items` without specifying `region_code` or `order_item_id`. - Run any ETL or BI query that touches the new columns to ensure no `NULL`?related errors.
4	Primary?key / uniqueness of `order_item_id`	If the column is intended as a key, existing rows lacking a value will violate the constraint.	- Check if a unique constraint or PK was added: `SHOW CONSTRAINTS IN order_items;` - If a PK exists, run an `UPDATE` that generates surrogate keys for all rows.
5	Join?logic with new `customers` table	Any query that joins `orders` or `order_items` to `customers` must handle missing rows (e.g., `LEFT JOIN`).	Run representative joins: `SELECT  FROM orders o LEFT JOIN customers c USING (customer_id) LIMIT 10;`
6	Data?type compatibility in downstream code	Stored procedures, views, scheduled SQL files, and ETL jobs may have hard?coded type assumptions.	- Run the entire suite of unit tests / integration tests that exercise these objects. - Use a "dry?run" of ETL scripts on the staging database and capture any type?cast or overflow errors.
7	Performance regression	The new column types might affect query plans (e.g., `TIMESTAMP_NTZ` vs `DATE`).	- Measure query latency before and after the migration on a representative workload. - Review the `EXPLAIN` plans for the top 10 slow queries.
8	Back?port / rollback strategy	In case something fails, you need a quick path back to v1.	Document the exact DDL used to create v2. Prepare a reverse migration script that drops new columns

Engineering counterpart
7:21 AM - Responded in 0s
What are the top risks, and what's the rollback plan if this doesn't work?

Agent response
7:21 AM - Responded in 26s

1.?Top Risks (ordered by severity and likelihood)

Rank	Risk	What could go wrong	Potential impact
A	`orders.order_date` type mismatch	Existing `DATE` values are stored in a column that Snowflake now expects to be `TIMESTAMP_NTZ`. Down?stream queries that cast or assume a `DATE` will fail or return wrong results (e.g., `ORDER BY order_date` may treat midnight as the same value but `WHERE order_date = '2024?03?01'` may raise a cast error).	Data quality - incorrect dates, broken reports; runtime errors in ETL pipelines.
B	Missing values for `order_items.order_item_id`	If the new column is defined as `NOT NULL` or is part of a PK, any pre?existing rows will violate the constraint, causing inserts/updates to error out.	ETL failure - batch jobs may abort; data loss or inconsistent state.
C	Join logic assumptions	Code that expects a `customers` table or a `region_code` in `orders` may return `NULL` values unexpectedly, leading to downstream logic errors or mis?attributed analytics.	Business?critical reports may show wrong region/segment information.
D	Performance regression	The new timestamp type may cause larger I/O or different clustering behavior, making previously fast queries slower.	User?experience degradation; increased compute cost.
E	Untracked constraints / indexes	If the migration script added PKs, FKs, or clustering keys, they may not be present in the snapshot you're relying on, leading to unnoticed integrity issues.	Data integrity violations, duplicate orders, etc.

2.?Rollback Plan (if the migration fails or any risk materializes)

> Goal: Re?establish the exact 2024?Q1 catalog state and restore all data to the pre?migration condition, without data loss or corruption.

2.1.?Prerequisites

Action	Why it matters
Create a full database snapshot (or clone) after the migration but before any data changes.	Provides a quick fallback if you need to recover data that may have been altered during rollback.
Document the exact DDL used to create v2 (including any `ALTER TABLE` statements).	Needed to generate the reverse DDL.
Verify that all user?defined objects (views, stored procedures, streams, tasks, etc.) still reference the same tables.	Some objects might need to be dropped/recreated if their definitions depend on the new schema.

2.2.?Rollback Steps

> Assume you are working in a staging environment first; once confirmed, repeat in production.

1. Drop new objects (if any)
   ```sql
   DROP TABLE IF EXISTS customers;
   ```

2. Revert `orders.order_date` type
   ```sql
   ALTER TABLE orders ALTER COLUMN order_date SET DATA TYPE DATE;
   ```

   Snowflake will automatically convert `TIMESTAMP

Engineering counterpart
7:21 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.