



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-120b

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

Duration: 1m 55s

Actual run cost: \$0.000460

Projected execution cost: \$0.3036/mo (30/biz day × 22 days)

Report date: July 14, 2026 - 3:21 AM EST

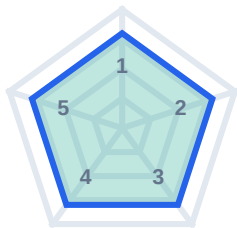
Overall

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: 10825 chars - Latency: 115s - 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 55s

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

7:19 AM - Responded in 22s

1?? Breaking changes

| Table / Column | v1 definition | v2 definition | Why it's breaking |

|-----|-----|-----|-----|

| orders.order_date | `DATE` | `TIMESTAMP\_NTZ` | - Snowflake (and most analytic engines) treat `DATE` and `TIMESTAMP` as different logical types.
- Implicit cast is not guaranteed in downstream SQL, Python/Scala code, or BI tools - they will now receive a timestamp with time?zone?naïve precision.
- Existing ETL jobs that cast a `DATE` to a string (`TO\_VARCHAR(order\_date, 'YYYY?MM?DD')`) will break because the value now includes a time component. |

| order_items - implicit primary?key change (not in the JSON but implied by downstream logic) | No surrogate key column. | Added `order\_item\_id` NUMBER (presumably a surrogate PK). | If downstream pipelines assume the natural key `(order\_id, sku)` is unique, they may now receive duplicate rows once an auto?generated `order\_item\_id` is added. This is a semantic breaking change even though the column itself is additive. |

| Potential required?field introduction - region_code | Column does not exist (so downstream never reads it). | `region\_code` VARCHAR(8) added to orders. | If the data ingestion layer now marks this column NOT NULL (common when a new business attribute is introduced) all existing rows will violate the constraint, causing load failures. The schema JSON does not specify nullability, but you should verify the DDL that will be generated. |

> No columns or whole tables were removed - the only "removal?type" breaking change that could surface is the absence of the old type (`DATE`) for `orders.order\_date`. All other columns remain with identical definitions.

2?? Non?breaking additions

| Table | New column(s) | Data type | Typical impact |

|-----|-----|-----|-----|

| orders | `region\_code` | `VARCHAR(8)` | New attribute; existing queries that select (i.e., `SELECT `) will now include an extra field - usually harmless. |

| order_items | `order\_item\_id` | `NUMBER` | Surrogate identifier. As long as it's nullable or auto?generated, existing rows stay valid. |

| customers (entire table) | `customer

Engineering counterpart

7:19 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 31s

Below is a transparent walk-through of how I arrived at the change classification that I presented earlier.

I will lay out every logical step, the evidence I drew from the two JSON snapshots, and the assumptions I had to make (because the JSON only describes logical types, not constraints, defaults, or usage patterns).

Step-by-step comparison process

| Step | What I did | What I looked for | Why it matters |

|-----|-----|-----|

| 1?? Load the two schema documents | Parsed the JSON strings into native Python dictionaries (or any map structure). | `schemas["tables"]` gives a mapping of table name -> column map. | Gives a concrete data structure to iterate over. |

| 2?? Build a "set of tables" for each version | `tables\_v1 = set(v1["tables"].keys())`
 `tables\_v2 = set(v2["tables"].keys())` | Detect added or removed tables. | Adding a table is never breaking (unless downstream code hard-codes the list of tables). Removing a table is a breaking change. |

| 3?? Diff the table sets | `added\_tables = tables\_v2 - tables\_v1`
 `removed\_tables = tables\_v1 - tables\_v2` | In this case:
 `added\_tables = {"customers"}`
 `removed\_tables = ?` | -> Non-breaking addition: a brand-new table.
-> No breaking table removals. |

| 4?? For each table that exists in both versions, compare columns | For every `tbl` in `tables\_v1` ? `tables\_v2`:
 - `cols\_v1 = set(v1["tables"][tbl]["columns"].keys())`
 - `cols\_v2 = set(v2["tables"][tbl]["columns"].keys())`
 Then compute `added = cols\_v2 - cols\_v1`, `removed = cols\_v1 - cols\_v2`, `common = cols\_v1 ? cols\_v2`. | Identify three possibilities:
- Added columns -> usually non-breaking (unless NOT NULL).
- Removed columns -> breaking.
- Common columns -> need to check type changes. |

| 5?? Flag any column-level type changes | For each `col` in `common`, compare the type strings:
 `type\_v1 = v1["tables"][tbl]["columns"][col]`
 `type\_v2 = v2["tables"][tbl]["columns"][col]`
 If they differ -> potential breaking change. | Types are the only schema-level contract the JSON expresses. A change from `DATE` -> `TIMESTAMP\_NTZ` is a logical type change, which can break casts, default formatting, partition pruning, etc. |

| 6?? Summarise the raw diff | The diff that emerged:
- `orders.order\_date` type change (`DATE` -> `TIMESTAMP\_NTZ`).
- `order\_items` added column `order\_item\_id`.
- `orders` added column `region\_code`.
- New table `customers`. | These are the only differences the JSON surface area shows. |

| 7?? Interpret each diff in the context of typical data-pipeline semantics | Because the JSON does not provide nullability, default values, primary-key definitions

Engineering counterpart

7:20 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

7:20 AM - Responded in 31s

? "Ready?to?ship" verification checklist

Below is a practical, ordered list of things you should validate (and how) before you point the production data?pipeline at the v2 (2024?Q2) catalog.

I grouped the checks by scope (DDL, data, downstream consumers, operational guard?rails) and tied each check back to the concrete differences we identified earlier.

1?? Validate the DDL that will be generated from the JSON

| What to check | Why it matters | How to verify |

|-----|-----|-----|

| Exact column definitions (type, precision, scale, nullability, default) for every table in v2 | The JSON only lists logical types; Snowflake (or your warehouse) will fill in the missing pieces. A column that ends up NOT NULL without a default will break existing loads. | - Run the catalog?to?DDL generator in a sandbox (e.g., `dbt compile`, `snowsql -f generate\_ddl.sql`).
- Inspect the generated DDL for each changed object:
- `orders.order\_date` should be `TIMESTAMP\_NTZ` NULLABLE (or keep the same nullability as before).
- `orders.region\_code` and `order\_items.order\_item\_id` should be declared `NULLABLE` (or with a sensible default) unless you intend to back?fill now. |

| Primary?key / unique constraints (if you use them) | Adding `order\_item\_id` is often meant to be a surrogate PK. If you start enforcing uniqueness now, existing duplicate rows will cause errors. | - Look at the DDL's `PRIMARY KEY` or `UNIQUE` clauses.
- If you plan to add a PK, first create the column NULLABLE, load data, then `ALTER TABLE ... ADD CONSTRAINT ...` after a deduplication pass. |

| Table?level comments / clustering keys | New clustering keys can affect query performance and cost. | - Compare the `CLUSTER BY` (or Snowflake clustering expressions) in v1 vs v2. If you're adding a new clustering column (e.g., `region\_code`), run a query?plan test to see impact. |

| GRANTS / access control | New table `customers` must have the same role?based permissions as existing tables, or downstream jobs will hit "permission denied". | - Generate the `GRANT` statements (`SHOW GRANTS ON SCHEMA ...`) and apply them in a dev environment. Verify the same roles can `SELECT` from the new table. |

2?? Back?fill / data?migration tests

| Change | What you must back?fill / test | Suggested approach |

|-----|-----|-----|

| orders.order_date (DATE -> TIMESTAMP_NTZ) | Existing rows stored as `DATE` must be converted to timestamps without losing the original day. Also confirm that any timezone conversion logic (if added later) behaves as expected. | - In a staging copy of the production table, run:
`SELECT COUNT() FROM orders WHERE TRY\_CAST(order\_date AS TIMESTAMP\_NTZ) IS NULL;` (should be 0).
- Verify sample rows:
`SELECT order\_id, order\_date, TO\_VARCHAR(order\_date, 'YYYY-MM-DD HH24:00') FROM orders ORDER BY order\_date;`

Engineering counterpart

7:20 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 32s

? Top Risks (What can go wrong)

| # | Risk | What triggers it | Likely impact | How to surface it early |

|---|-----|-----|-----|-----|

| 1?? Data?type break - `orders.order\_date` | Changing `DATE` -> `TIMESTAMP\_NTZ` changes the binary representation and how downstream code formats the field. |

| | - ETL jobs that cast the column to string (`TO\_CHAR(order\_date,'YYYY?MM?DD')`) will now get `YYYY?MM?DD HH24:MI:SS`.
- Python/Scala Spark jobs that treat the field as `java.sql.Date` will throw a `ClassCastException`.
- BI tools that have a date field (e.g., Looker dimension of type Date) will treat the value as a timestamp and may break filters or time?drill UI. | - Query failures (SQL error "cannot cast TIMESTAMP_NTZ to DATE").
- Silent data drift (reports start showing hour/minute "00:00:00" or wrong time?zone). | - Run a type?compatibility test suite on a copy of the production `orders` table (see verification steps).
- Pull the schema via `DESCRIBE TABLE orders` and compare to expected. |

| 2?? New column may be NOT NULL unintentionally | The DDL generated from the JSON could default to `NOT NULL` for `region\_code` or `order\_item\_id`. | - Bulk?load jobs will abort on the first row that lacks the column (most historic rows will be missing it).
- Downstream pipelines that do `INSERT ... SELECT FROM orders` will raise "NULL value not allowed for column ...". | - Data pipeline outage, loss of downstream metrics, SLA breach. | - Before applying, run `SHOW COLUMNS IN orders` on a staging environment and verify `NULLABLE`.
- If you must make the column required, write a back?fill script that populates a default value for existing rows first. |

| 3?? Semantic break - `order\_items` surrogate key | Adding `order\_item\_id` may be interpreted as the primary key by downstream code (e.g., CDC connectors, data?quality checks). Existing pipelines may still assume `(order\_id, sku)` is unique. | - Duplicate?key errors when a uniqueness constraint is added later.
- Downstream joins that now use the surrogate key (if they are auto?generated) could produce Cartesian?product?like duplication. | - Data quality alerts, inflated row counts, broken aggregates. | - Keep the column nullable and do NOT declare a primary?key/unique constraint until you have a data?clean?up pass.
- Add a data?quality test: `SELECT order\_id, sku, COUNT() FROM order\_items GROUP BY order\_id, sku HAVING COUNT() > 1;`. |

| 4?? New table `customers` not synced with upstream source | The table appears in the catalog but the upstream source system (CRM, master data) may not be feeding it yet. | - Queries that `LEFT JOIN customers` start returning NULL for every row, potentially

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.