



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: inclusionai/ling-2.6-flash

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

Duration: 18s

Actual run cost: \$0.000102

Projected execution cost: \$0.0673/mo (30/biz day × 22 days)

Report date: July 13, 2026 - 11:10 PM EST

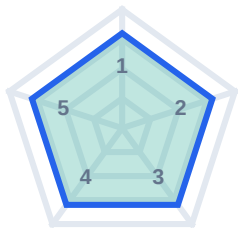
Overall

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: 12936 chars - Latency: 18s - 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: 18s

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

Engineering counterpart

3:09 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

3:10 AM - Responded in 4s

Below is a concise, column- and table-level migration impact review based strictly on the provided JSON catalog snapshots.

---

### ## 1) Breaking Changes

#### ### Table and Column Type/Definition Changes

##### - orders.order\_date

- v1: `DATE`

- v2: `TIMESTAMP\_NTZ`

- Impact: Type change may break time-zone-sensitive comparisons, date truncation logic, and applications performing direct date comparisons or relying on DATE-specific behavior.

##### - order\_items

- Column removed: `order\_items.order\_id` (sole key in v1)

- Column added: `order\_items.order\_item\_id` (new primary key in v2)

- Impact: Existing application queries and ETL jobs referencing `order\_items.order\_id` will fail at runtime. Applications expecting exactly one row per order item may also break if `order\_item\_id` is not populated correctly.

#### ### New Required Fields / Constraints Cannot Be Verified from Schema Alone

- No indication of new NOT NULL constraints or primary key changes (e.g., whether `order\_item\_id` is required), but the structural shift from `order\_id` to `order\_item\_id` as the identifier is significant.

---

### ## 2) Non-Breaking Additions

#### ### New Tables

##### - customers

- Columns: `customer\_id`, `region`, `segment`

- Status: New table; safe addition if properly backfilled and if applications are updated to use it.

#### ### New Columns (Non-Breaking When Optional)

##### - orders.region\_code (`VARCHAR(8)`)

- Optional addition; applications not using it will be unaffected.

---

### ## 3) Migration Risks and Recommended Verification Steps

#### ### High-Risk Items

##### 1. orders.order\_date type change (DATE -> TIMESTAMP\_NTZ)

- Risk: Query results, date math, and index behavior may change; potential for silent data truncation or misinterpretation if sessions/DB time zones differ.

- Verification:

- Run core query tests that filter, group, and cast `order\_date`.

- Validate date math and comparisons still perform as expected.

- Confirm downstream BI/reporting tools handle TIMESTAMP\_NTZ semantics.

##### 2. order\_items identity change (order\_id -> order\_item\_id)

- Risk: Foreign-key relationships and application join logic break if not updated.

- Verification:

- Confirm referential integrity: ensure `order\_items.order\_id` in v1 maps correctly to the new keying strategy (or a

retained foreign key to orders).

- Validate ETL pipelines and application code that join or reference `order\_items.order\_id`.
- Check that `order\_item\_id` values are unique, non-null, and stable.

### 3. Missing orders.order\_id in order\_items (v1 key)

- Risk: Loss of direct order-item linkage at the column level; possible data model mismatch.
- Verification:
  - Ensure `order\_items.order\_id` is retained as a foreign key (even if not the primary key) or confirm the model intentionally removes it.
  - Validate joins between orders and order\_items via `order\_id` still work.

### ### Medium-Risk / Operational Items

- customers table addition
  - Risk: Applications reading from orders/order\_items may expect customer data in a joined shape and now require an explicit join to the new table.
  - Verification:
    - Confirm `customer\_id` exists in customers and matches orders.customer\_id.
    - Validate region/segment mappings and backfill completeness.

### ### Recommended Verification Checklist

- Run an end-to-end order + order items flow (insert -> select ->



**Engineering counterpart**  
**3:10 AM - Responded in 0s**

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

## Agent response

3:10 AM - Responded in 5s

Step-by-step reasoning and stated assumptions

### 1) Identify scope and goal

- I was asked to compare two JSON catalog snapshots (v1/Q1 and v2/Q2) and report:
  - Breaking changes
  - Non-breaking additions
  - Migration risks and verification steps
- Constraints: be specific (cite tables/columns), do not invent columns not present in the inputs.

### 2) Parse the schemas carefully

- v1 tables: orders, order\_items
- v2 tables: orders, order\_items, customers
- Column-level changes identified:
  - orders.order\_date: DATE -> TIMESTAMP\_NTZ
  - order\_items: lost order\_id as a column, gained order\_item\_id as a new column (order\_id is absent in v2 items)
  - orders.region\_code added
  - customers added with customer\_id, region, segment

### 3) Classify breaking vs non-breaking

- Type change on order\_date: DATE vs TIMESTAMP\_NTZ can affect query behavior, time zone handling, and downstream tools - classified as breaking.
- Key change in order\_items:
  - v1 primary/composite key context implied by "order\_id" alone in items.
  - v2 replaces order\_id with order\_item\_id as the new identifier column. Any SQL, code, or ETL that references order\_items.order\_id will fail at runtime (object no longer exists) - breaking.
  - Absence of order\_id in v2 items removes the direct column-level link; this is breaking to joins and referential assumptions unless order\_id is retained as a foreign key not shown in columns listed. Since columns listed do not include order\_id in v2 items, it's a breaking removal.
- Added required NOT NULL constraints cannot be inferred from the JSON (no nullability/key metadata), so I did not assume them - noted as unknown.
- New tables (customers) and new optional columns (orders.region\_code) are non-breaking if properly handled in application code - classified as non-breaking additions.

### 4) Map migration risks and verification

- High risks:
  - Type change can break date logic and BI.
  - Key change breaks joins and application queries.
  - Missing order\_id in items breaks referential integrity unless handled at application level.
- Recommendations:
  - Test date comparisons, truncations, and filters.
  - Validate join behavior between orders and order\_items using the new keying.
  - Confirm presence/absence of order\_id as a foreign key in v2 (not visible in columns) and update application code and ETL accordingly.
  - Backfill and verify customers table completeness and joins.

### 5) Explicit assumptions called out

- I assume the column lists are exhaustive for each table as provided.
- I assume no additional metadata (e.g., primary key flags, nullability, default values) beyond the column types shown.
- I assume "removal" of order\_items.order\_id is breaking because the column is absent in v2; I do not assume it is retained as a foreign key not listed.
- I do not invent columns beyond those in the inputs.

This reasoning leads to the summarized breaking changes, non-breaking additions, and verification steps in the previous

answer.

**Engineering counterpart**  
**3:10 AM - Responded in 0s**

What would you verify before shipping this to production?

## Agent response

### 3:10 AM - Responded in 4s

Before shipping to production, verify the following (based strictly on the schema differences and common production readiness checks):

#### Schema and model verification

- Confirm presence and data type of every column in v2:
  - orders: order\_id, customer\_id, order\_date (TIMESTAMP\_NTZ), total\_usd, status, region\_code
  - order\_items: order\_item\_id, order\_id (if retained as a foreign key), sku, qty, unit\_price
  - customers: customer\_id, region, segment
- Verify primary keys and uniqueness:
  - orders: confirm order\_id PK
  - order\_items: confirm order\_item\_id is PK and is populated correctly; if order\_id is retained, decide whether it is a foreign key or deprecated
- Verify nullability and defaults:
  - Determine which columns are nullable, and whether order\_items.order\_id is nullable or required
- Verify referential integrity and constraints:
  - Ensure foreign-key relationships are defined (orders.customer\_id -> customers.customer\_id; order\_items.order\_id -> orders.order\_id if retained)
  - Check ON DELETE/UPDATE rules

#### Data migration and backfill

- Plan and test backfill for customers (region, segment) and for any derived/computed columns
- Ensure order\_items has a reliable mechanism to populate order\_item\_id (sequence, auto-increment, or application logic) and that it is unique and stable
- Backfill region\_code in orders with correct, verifiable values and confirm defaults or migration job success rates

#### ETL, pipelines, and application code

- Update all ETL jobs, SQL queries, and application code to use order\_item\_id instead of order\_items.order\_id
- Validate join logic between orders and order\_items (whether using order\_id as FK or not)
- Verify any data transformation that depends on order\_date behavior with TIMESTAMP\_NTZ (filters, date arithmetic, time zone handling)
- Test read/write paths for the new customers table and any new joins or aggregations involving region/segment

#### Query and index performance

- Assess query plans for key reports and confirm indexes support new types and new joins (e.g., on customers.region/segment)
- Monitor performance of TIMESTAMP\_NTZ operations versus previous DATE behavior (index usage, partitioning)

#### Integrity and reconciliation checks

- Row counts: orders, order\_items, customers; ensure referential counts make sense (e.g., orphaned order\_items = 0)
- Sum/check totals: total\_usd aggregates, qty and unit\_price reconciliations
- Validate region\_code values are within expected set and consistent with any regional logic

#### Functional and regression testing

- End-to-end order creation, modification, cancellation flows
- Reporting and analytics queries involving order\_date, region\_code, customers.region/segment
- API/contract tests if schema is exposed via services

#### Operational readiness

- Rollback plan and data migration script tested in staging
- Monitoring and alerts for failed migrations, type conversion errors, or constraint violations
- Access control and permissions for new table/customers if needed

#### Communication and documentation

- Update data dictionary and schema docs to reflect order\_item\_id as new key and order\_id removal in order\_items
- Notify downstream consumers of date type change and key change

If any of the nullable/referential details are not specified in the catalog (as with v2), treat them as unknown and validate explicitly in staging before production promotion.

**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 4s**

Top risks (rooted in the catalog changes)

- Key breakage in order\_items: v1 used order\_id as the item identifier; v2 replaces it with order\_item\_id and omits order\_id. If applications or ETL still reference order\_items.order\_id, runtime errors will occur and joins between orders and order\_items will break.
- Type change on order\_date: DATE -> TIMESTAMP\_NTZ can alter query behavior, date truncation, and time-zone handling, causing silent mismatches or filter errors in reports and application logic.
- Missing foreign key on order\_items.order\_id (if intentionally removed): orphaned records or broken referential integrity between orders and order\_items if the link is not preserved or enforced at the application layer.
- New nullable columns (region\_code, customers.region/segment): if downstream code assumes NOT NULL or expects a fixed set of values, nulls can cause errors or bad aggregations.
- Data quality/backfill issues: customers or region\_code backfill may be incomplete or incorrect, leading to inconsistent reporting and joins.

Rollback plan (to restore prior state quickly and safely)

- Immediate stop
  - Pause any downstream jobs, pipelines, and application deploys that consume the new schema.
- Application/code rollback
  - Revert application code and ETL to use order\_items.order\_id (v1 behavior) or the prior interface; ensure feature flag or config rolls back to pre-Q2 endpoints/queries.
- Database rollback options
  - If the catalog and underlying DB changes are controlled by migration scripts, revert the migration scripts and run a compensating migration:
    - Restore order\_items: ensure order\_id is retained as a column or re-add it as a nullable column and backfill values from order\_item\_id where mapping exists.
    - Restore orders.order\_date to DATE if the TIMESTAMP\_NTZ change was part of an applied migration; otherwise ensure DATE semantics are maintained.
    - Drop region\_code from orders and remove customers table if it was introduced in the same migration batch (only if it does not impact production data required by other services).
- Data reconciliation
  - Re-sync any transformed data (e.g., orders.total\_usd, order\_items pricing/qty) by replaying write-ahead logs or change data capture from the pre-Q2 window if feasible.
  - Run reconciliation jobs: counts, sums, and key-join integrity between orders and order\_items.
- Cut back over feature flagging or routing
  - If the new schema is behind a feature flag or read-replica, flip traffic back to the stable environment/version.
- Validation after rollback
  - Run smoke tests for order creation, order\_items CRUD, and key reports.
  - Confirm no job failures or null-reference errors in application logs.
- Communication
  - Notify stakeholders of rollback and expected impact window; schedule a post-mortem and a controlled reattempt after fixes.

Note: If the catalog changes are already applied at the DB schema level (e.g., ALTERs already executed), a full rollback may require a backup restore to a pre-Q2 state; coordinate with DBA and ensure backups are verified and recover point objectives (RPO) and recovery time objectives (RTO) are met before proceeding.

**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.