



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/gemma-3-12b-it

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

Duration: 51s

Actual run cost: \$0.000510

Projected execution cost: \$0.3366/mo (30/biz day × 22 days)

Report date: July 13, 2026 - 11:08 PM EST

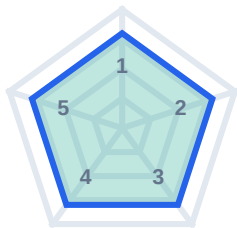
Overall

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: 13956 chars - Latency: 51s - 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: 51s

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

3:08 AM - Responded in 12s

Okay, here's a detailed review of the schema migration from v1 (2024-Q1) to v2 (2024-Q2), outlining breaking changes, non-breaking additions, associated risks, and recommended verification steps.

1. Breaking Changes

``orders.order_date`` Type Change:

v1: ``DATE``

v2: ``TIMESTAMP_NTZ``

Impact: This is a significant breaking change. Existing code likely assumes ``order_date`` represents just the date component. Converting to ``TIMESTAMP_NTZ`` introduces a time component. Applications might need to be updated to handle the time portion, particularly if the time component is not relevant to the business logic. Data type mismatches will also occur if querying or processing data in a system that expects ``DATE``.

New Required Field: ``orders.region_code``:

v1: Absent

v2: ``VARCHAR(8)``

Impact: This is a hard breaking change if the new column ``region_code`` is enforced as ``NOT NULL`` in the database. Existing data in the ``orders`` table will be missing this column's value, which will likely lead to errors when attempting to load or query the data using the new schema. Even if it is nullable, the absence of a value will need to be handled by the application.

Column Renaming: ``order_items.order_id`` to ``order_items.order_item_id``:

v1: ``order_id``

v2: ``order_item_id``

Impact: This is a breaking change. All queries, transformations, and code that references ``order_items.order_id`` will need to be updated to use ``order_items.order_item_id``.

2. Non-Breaking Additions

``orders.region_code`` (No impact on existing queries): The addition of this column doesn't break existing queries, as it's a new field. However, existing queries won't leverage this field until adapted.

``customers`` Table:

v1: Absent

v2: Present

The addition of the ``customers`` table itself doesn't break anything in the existing ``orders`` or ``order_items`` tables. Applications will simply need to be extended to utilize this new table, and it will not affect existing queries.

``customers.region``: New column in the newly introduced customer table.

``customers.segment``: New column in the newly introduced customer table.

3. Migration Risks and Recommended Verification Steps

Risk: Data Loss/Corruption due to Type Mismatch (``orders.order_date``)

Verification:

1. Data Sampling: Select a representative sample of ``orders`` records from both v1 and v2. Visually inspect the ``order_date`` values. Verify that the time component in v2 is handled appropriately by any downstream processes.

2. Query Test: Run queries that filter or sort by ``order_date`` in both schemas. Confirm the results are logically consistent, especially around date boundaries.

3. Transformation Impact: Identify all ETL processes or data pipelines that transform or use the ``order_date`` column. Review and test these transformations after the migration to ensure they correctly handle the ``TIMESTAMP_NTZ`` data type.

Risk: Application Errors Due to

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

Agent response

3:08 AM - Responded in 14s

Okay, let's walk through my reasoning step-by-step for the schema migration review, including the assumptions I made. I'll break it down into how I approached each part of the analysis: breaking changes, non-breaking additions, and then the risks/verification.

1. Overall Approach & Initial Assumptions

Goal: My primary goal was to identify changes that would actively break existing code or data processing pipelines, and changes that are additive without causing immediate breakage. I assume we want to minimize disruption and ensure data integrity.

Schema Definition: I'm treating the JSON schemas as defining the structure of the data. I'm assuming this schema represents the actual data types and constraints enforced within the underlying database or data store. If the schemas are purely descriptive and don't reflect enforced constraints, the risk assessment would need to be adjusted.

"Breaking" Definition: I'm using a fairly strict definition of "breaking". A change is breaking if it requires modification to existing code, queries, or data pipelines to function correctly. A nullable column that's suddenly required is a breaking change.

No Knowledge of Existing Code: I proceeded without knowing the specifics of the applications or pipelines that use this data. This means I'm looking for the potential for breakage, not necessarily guaranteed breakage.

2. Identifying Breaking Changes - Step by Step

Comparing Tables & Columns: I systematically compared each table and column between v1 and v2.

`orders` Table: I started by looking for differences in the `columns` section.

`order\_date`: I immediately noticed the type change from `DATE` to `TIMESTAMP\_NTZ`. This is a strong indicator of a breaking change. My reasoning was that date-specific logic in existing code will likely fail or produce incorrect results when it encounters a timestamp.

`region\_code`: The new column flagged as a potential breaking change. If this column is required (i.e., `NOT NULL` in the underlying database), existing data will fail to load or be queried. Even if it's nullable, existing code might not be prepared to handle empty or default values.

`order\_items` Table:

`order\_id` to `order\_item\_id`: This renaming is a clear breaking change. References to the old column name will simply fail. It's also relatively easy to identify because it's a direct name change within the same table.

No other changes within `orders` or `order\_items`: The remaining columns were identical.

3. Identifying Non-Breaking Additions - Step by Step

New Tables: I looked for tables present in v2 but absent in v1.

`customers`: The entirely new table immediately stood out as a new addition. The absence of this table in v1 means that its addition cannot break existing queries or code that only operates on `orders` or `order\_items`.

New Columns within Existing Tables: I looked for new columns in existing tables.

`orders.region\_code`: While new, this column doesn't break anything if it's nullable, because existing queries won't be affected if this column is not explicitly used.

`customers.region` and `customers.segment`: These are new columns within a new table. Similar to the `customers` table itself, they are non-breaking.

4. Assessing Risks and Recommended Verification Steps - Step by Step

What would you verify before shipping this to production?

Agent response

3:08 AM - Responded in 13s

Okay, assuming this schema migration review has been completed and documented, here's a detailed checklist of what I'd verify before shipping this to production. This is broken down into categories: Data Validation, Application/Code Testing, and Infrastructure/Deployment Considerations. I'll also include a risk prioritization within each.

I. Data Validation (High Priority - Absolute Must)

Data Completeness After Migration:

Verification: Run a COUNT() query on each table in both the source (v1) and target (v2) schemas after the migration has been applied to a staging or pre-production environment. The counts must match (or be very close, accounting for any data cleansing/transformation during the migration).

Rationale: Confirms no data loss during the migration process.

Data Type Validation - `orders.order\_date`:

Verification: Sample a significant number (e.g., 1000) of `orders` records from both v1 and v2. Specifically compare the `order\_date` values. Ensure the new timestamp values are logically consistent with the original date. Check for unexpected time components if they weren't intended. This requires a manual or scripted comparison.

Rationale: Catches potential data corruption or mishandling of timestamps.

Nullability Checks - `orders.region\_code`:

Verification: Query the target schema for all `orders` records where `region\_code` is NULL. Assess what the business behavior should be in this situation - is a default code expected? Does the application need to handle nulls?

Rationale: Ensures graceful handling of missing data in the new column.

Referential Integrity Checks:

Verification: If `order\_id` in `order\_items` is used as a foreign key referencing `orders.order\_id`, ensure the relationship is still valid after the column rename. Run queries to confirm all `order\_items` records have corresponding `orders` records.

Rationale: Prevents orphaned records and data inconsistencies.

II. Application/Code Testing (High to Medium Priority - Crucial)

End-to-End Functional Testing:

Verification: Execute ALL critical business workflows that use data from these tables. This includes:

- Order creation
- Order retrieval (by various criteria - date, customer, region, etc.)
- Order modification
- Reporting on orders and order items
- Customer data retrieval & updates

Rationale: Uncovers integration issues that unit tests might miss.

Query Performance Testing:

Verification: Run representative queries (including those that now utilize `orders.region\_code`) against the target schema. Compare performance to the source schema. Identify and address any performance regressions.

Rationale: Schema changes can impact query performance.

Code Modification Verification (specifically around column renaming):

Verification: Automated tests targeting areas where `order\_items.order\_id` was previously used. These tests should now use `order\_items.order\_item\_id`. Static code analysis tools can also help with this.

Rationale: Confirms the column rename has been correctly implemented in the codebase.

Application Logging & Error Handling:

Verification: Review application logs for any errors or unexpected behavior related to the schema changes. Ensure error handling is robust enough to deal with potential issues.

III. Infrastructure/Deployment Considerations (

Engineering counterpart
3:08 AM - Responded in 0s

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

Agent response

3:08 AM - Responded in 13s

Okay, let's focus on the top risks associated with this schema migration and a detailed rollback plan.

Top Risks (Ranked by Severity)

1. Data Corruption/Loss due to `order\_date` Type Mismatch (Critical): This is the highest risk. Incorrect handling of the `TIMESTAMP\_NTZ` data type can lead to inaccurate reporting, broken downstream processes, and a loss of confidence in the data. A subtle bug here could be extremely difficult to debug.
2. Application Failure due to `order\_items.order\_id` Rename (High): If the code changes to reflect the new column name (`order\_item\_id`) aren't complete or are implemented incorrectly, key application functionality will fail. This could manifest as order processing errors, reporting issues, or even system crashes.
3. Data Loading/ETL Pipeline Failures due to `orders.region\_code` (Medium-High): If the new `region\_code` column isn't handled appropriately (e.g., missing default values, application not expecting it), data loading processes can fail, leading to incomplete or inaccurate data in the target system.
4. Performance Degradation (Medium): While not immediately catastrophic, performance regressions after the schema change can impact user experience and system scalability. Slow queries can become a major operational issue.
5. Unexpected Application Behavior (Low-Medium): Residual code that hasn't been updated to handle the new schema might exhibit unexpected behavior. This is less likely if thorough testing is performed, but still a possibility.

Rollback Plan (Detailed and Prioritized)

The overall rollback strategy needs to be reversible, meaning we can return to the original schema (v1) with minimal data loss and downtime. The plan has several layers:

Phase 1: Immediate Rollback (Within Minutes - Emergency)

Trigger: If, within the first few minutes of production deployment, we observe critical errors (e.g., order processing failures, severe query performance degradation, data integrity issues) that are clearly linked to the schema change.

Action:

1. Stop All ETL Processes: Immediately halt any data pipelines that are writing to the new schema. This prevents further data corruption.

2. Switch Back to v1 Schema: This is the trickiest part and depends heavily on the database technology. Possible methods:

Database Schema Versioning (Ideal): If your database supports schema versioning, revert to the previous (v1) schema version. This is the cleanest and fastest option.

Backup and Restore: Restore the database to a backup taken before the schema migration. This is more disruptive but reliable.

Schema Rewind (Complex): If possible and the database allows, attempt a "schema rewind" operation. This would involve reverting the column type change for `order\_date`, removing the `region\_code` column, and renaming `order\_item\_id` back to `order\_id`. This is highly complex and potentially risky.

3. Verify Rollback: After reverting the schema, verify that critical applications are functioning correctly and that data integrity is restored. Run essential queries and confirm that data counts are as expected.

Phase 2: Controlled Rollback (Within Hours - Planned)

Trigger: If the issues are less severe but still impacting the system (e.g., performance degradation, minor application glitches), or if the initial rollout highlights unforeseen challenges.

Action:

1. Pause New Data Ingestion: Stop all data pipelines to prevent further data inconsistencies.
2. Schema Reversion: Follow the same steps as in Phase 1

Engineering counterpart

3:08 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.