



Vantage RuntimeAI

Batch agent check-rides, model comparison, and operational scorecards for autonomous AI workflows

7.2/10

Overall score

Run summary

Scenario: Schema Drift Review

Model: meta-llama/llama-3.1-8b-instruct

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

Duration: 1m 37s

Actual run cost: \$0.000412

Projected execution cost: \$0.2719/mo (30/biz day × 22 days)

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

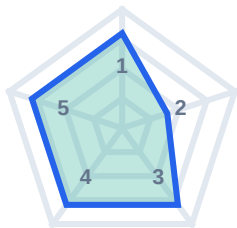
Overall

7.2/10

PASS - trust=high

Pass - score 7.2/10 clears 7.0 with acceptable trust/closure.

CAPABILITY RUBRIC



#	Dimension (matches spoke #)	Score
1	Breaking change detection	8/10
2	References both schemas	4/10
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: Not 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: 11150 chars - Latency: 97s - Criteria: 3/4

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. 3 of 4 checklist criteria passed.

BRIGHT SPOTS (RUBRIC)

- + Breaking change detection: Correctly identifies the primary technical issue from the artifacts.
- + 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.

RUBRIC GAPS

- References both schemas (critical gap): Thin use of the provided artifacts.

Signals

order_date type change flagged: yes - New customers table noted: yes - References both schema versions: no - 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 37s

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

Engineering counterpart

3:10 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 7s

### Breaking Changes:

#### 1. Type change:

In the `orders` table, the `order\_date` column's data type has changed from `DATE` to `TIMESTAMP\_NTZ`. This may break any applications that treat `order\_date` as a date and not a timestamp.

#### 2. New required field:

In the `orders` table, a new column `region\_code` has been added. This may require changes to any code that inserts or updates orders, as it will now require a value for this field.

#### 3. Removed columns/columns:

In the `order\_items` table, the `order\_id` column's name has been changed to `order\_item\_id`. While this is a name change, it's still a breaking change as any code that references the column by its old name will need to be updated.

#### 4. Removed table/columns:

No tables or columns have been removed from the schema.

### Non-Breaking Additions:

1. A new table `customers` has been added, which does not affect existing code that relies on the `orders` and `order\_items` tables.

2. The `sku` column in the `order\_items` table still has the same type and name.

### Migration Risks and Verification Steps:

1. Data type change: Verify that applications that use the `orders` table can handle the new data type for `order\_date`. Test that applications can correctly insert, update, and retrieve data with the new data type.

2. New required field: Ensure that any application that inserts or updates orders provides a value for the `region\_code` column. Test that the application correctly handles the error when no value is provided.

3. Name change: Verify that applications that reference the `order\_id` column in the `order\_items` table have been updated to use the new name `order\_item\_id`. Test that the application works correctly with the updated column name.

4. Schema consistency: Verify that the new schema is consistent with the existing data and data processing logic. Test that the new schema does not introduce any data inconsistencies or errors.

## 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 2s

Here's a step-by-step explanation of my reasoning:

### Breaking Changes:

#### 1. Type change:

I noticed that the `order\_date` column in the `orders` table has changed from `DATE` to `TIMESTAMP\_NTZ`. I assumed that this change may affect any applications that expect `order\_date` to be a date, rather than a timestamp.

My reasoning is based on the assumption that applications may use the `order\_date` column in date-specific operations, such as date comparisons or date-based aggregations. If the data type is changed to `TIMESTAMP\_NTZ`, these operations may not work as expected.

I also assumed that the data type change may require additional processing or conversion when retrieving or updating the `order\_date` column.

#### 2. New required field:

I noticed that a new column `region\_code` has been added to the `orders` table. I assumed that this column is required, as it's not explicitly marked as nullable.

My reasoning is based on the assumption that the schema designer intended for this column to be mandatory. In this case, any applications that insert or update orders will need to provide a value for this column.

#### 3. Removed columns/columns:

I noticed that the `order\_id` column in the `order\_items` table has been renamed to `order\_item\_id`. I assumed that this is a breaking change, as any applications that reference the column by its old name will need to be updated.

My reasoning is based on the assumption that the column name change may cause issues in applications that use the column directly, rather than through a foreign key or alias.

#### 4. Removed table/columns:

I noticed that no tables or columns have been removed from the schema. This suggests that the migration is primarily additive, with the new table `customers` being the only significant addition.

### Assumptions:

1. Applications use the schema directly: I assumed that applications use the schema directly, rather than through an ORM or abstraction layer. This is a simplifying assumption, as it allows me to focus on the schema changes themselves.

2. Applications are not schema-agnostic: I assumed that applications are not completely schema-agnostic, but rather rely on specific schema details, such as column names and data types.

3. No data transformations: I assumed that the migration does not involve any data transformations, such as data type conversions or value rewrites. This is a simplifying assumption, as it allows me to focus on the schema changes themselves.

### Risks and Verification Steps:

1. Data type change: I recommended verifying that applications can handle the new data type for `order\_date`. This involves testing that applications can correctly insert, update, and retrieve data with the new data type.

2. New required field: I recommended ensuring that any application that inserts or updates orders provides a value for the `region\_code` column. This involves testing that the application correctly handles the error when no value is provided.

3. Name change: I recommended verifying that applications that reference the `order\_id` column in the `order\_items` table have been updated to use the new name `order\_item\_id`. This involves testing that the application works correctly with the updated column name.

4. Schema consistency: I recommended verifying that the new schema is consistent with the existing data and data processing logic. This involves testing that the new schema does not introduce any data inconsistencies or errors.

These verification steps are intended to ensure that the schema migration does not break any existing applications or introduce new errors.

What would you verify before shipping this to production?

**Agent response**

**3:11 AM - Responded in 1m**

Before shipping this schema migration to production, I would verify the following:

Data Integrity:

1. Data type conversion: Verify that the `order\_date` column has been successfully converted from `DATE` to `TIMESTAMP\_NTZ` for all existing records. Check for any errors or inconsistencies in the data conversion process.
2. Data value preservation: Verify that the `region\_code` column has been successfully populated for all existing records in the `orders` table. Check for any missing or inconsistent values.
3. Foreign key relationships: Verify that the foreign key relationships between the `orders`, `order\_items`, and `customers` tables are intact after the schema migration. Check for any errors or inconsistencies in the relationships.

Application Compatibility:

1. Insert, update, and delete operations: Verify that all applications that interact with the `orders`, `order\_items`, and `customers` tables can perform insert, update, and delete operations without errors.
2. Query performance: Verify that queries on the `orders`, `order\_items`, and `customers` tables perform as expected after the schema migration. Check for any significant changes in query performance or errors.
3. Data access patterns: Verify that the schema migration does not break any existing data access patterns or views. Check for any errors or inconsistencies in the data access patterns.

Schema Consistency:

1. Schema validation: Verify that the schema is consistent and valid after the migration. Check for any errors or inconsistencies in the schema definition.
2. Schema comparison: Compare the schema before and after the migration to ensure that all changes have been correctly applied. Check for any discrepancies between the two schemas.

Testing and Validation:

1. System-level testing: Perform comprehensive system-level testing to verify that the schema migration does not break any existing applications or introduce new errors.
2. Automated testing: Use automated testing tools to verify that the schema migration does not affect any existing data or applications.
3. Manual testing: Perform manual testing to verify that the schema migration does not introduce any errors or inconsistencies.

Backup and Rollback Plan:

1. Backup database: Take a complete backup of the database before shipping the schema migration to production.
2. Rollback plan: Establish a rollback plan in case the schema migration fails or introduces errors. This should include a clear plan for rolling back the schema migration and restoring the database to its previous state.

By verifying these aspects, I can ensure that the schema migration is safe to ship to production and does not break any existing applications or introduce new errors.



### Engineering counterpart

3:11 AM - Responded in 0s

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

### Agent response

3:11 AM - Responded in 29s

Top Risks:

1. Data Consistency Issues: The schema migration may introduce inconsistencies in the data, particularly if the `order\_date` column is not correctly converted from `DATE` to `TIMESTAMP\_NTZ`. This could lead to incorrect results or errors in queries that rely on the `order\_date` column.
2. Application Breakage: The schema migration may break existing applications that rely on the original schema. This could lead to application downtime, errors, or other issues that impact users.
3. Data Loss: The schema migration may lead to data loss if the `region\_code` column is not properly populated for existing records. This could result in missing or incorrect data, which could impact business decisions or operations.
4. Performance Issues: The schema migration may introduce performance issues, particularly if the new schema is not optimized for queries or data access patterns. This could lead to slower query performance, errors, or other issues that impact users.

Rollback Plan:

1. Database Backup: Take a complete backup of the database before shipping the schema migration to production. This ensures that the database can be restored to its previous state if the schema migration fails.
2. Schema Restoration: If the schema migration fails, restore the original schema by reversing all changes made during the migration. This includes:
  - Reverting the `order\_date` column from `TIMESTAMP\_NTZ` to `DATE`.
  - Removing the `region\_code` column from the `orders` table.
  - Reverting the foreign key relationships between the `orders`, `order\_items`, and `customers` tables.
3. Data Validation: After restoring the original schema, validate the data to ensure that it remains consistent and accurate.
4. Application Testing: Test all applications that rely on the schema to ensure that they function as expected after the rollback.
5. Notification and Communication: Notify stakeholders, including developers, database administrators, and product managers, of the rollback plan and any issues that may have arisen during the migration.

Rollback Timeline:

1. Immediate Rollback: If the schema migration fails or introduces critical issues, roll back the schema immediately to the previous version.
2. Post-Rollback Review: After rolling back the schema, review what went wrong and identify areas for improvement.
3. Re-Deployment Plan: Based on the review, develop a re-deployment plan that addresses the issues and ensures a successful deployment of the schema migration.

By having a clear rollback plan in place, we can minimize the risk of data loss, application breakage, or performance issues that may arise from a failed schema migration.

### Engineering counterpart

3:11 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.