



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-4o-mini

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

Duration: 40s

Actual run cost: \$0.001740

Projected execution cost: \$1.15/mo (30/biz day × 22 days)

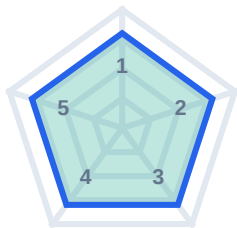
Report date: July 13, 2026 - 11:13 PM EST

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: 14751 chars - Latency: 40s - 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: 40s

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

Engineering counterpart

3:12 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

3:12 AM - Responded in 8s

### ### Breaking Changes

#### 1. Type Changes:

- In the `orders` table, the `order\_date` column has changed from type `DATE` in v1 to `TIMESTAMP\_NTZ` in v2. This is a breaking change as it alters the data type.

#### 2. New Required Fields:

- The `orders` table in v2 introduces a new column, `region\_code` of type `VARCHAR(8)`, which is not present in v1. If this column is required (not nullable), this would be a breaking change.
- The `order\_items` table in v2 introduces a new column, `order\_item\_id` of type `NUMBER`, which was not present in v1. If this column is required (not nullable), this also represents a breaking change.

#### 3. Removed Tables/Columns:

- No tables or columns have been removed in the transition from v1 to v2.

### ### Non-Breaking Additions

#### 1. New Table:

- The `customers` table has been added in v2, which includes `customer\_id`, `region`, and `segment` columns. This addition is non-breaking as it does not affect existing tables or relationships directly.

#### 2. Additional Columns:

- The `orders` table has an additional column `region\_code`, and the `order\_items` table has an additional column `order\_item\_id`. These additions could enhance the schema without breaking existing functionality.

### ### Migration Risks and Recommended Verification Steps

#### 1. Data Type Conversion Risks:

- Ensure that existing data in `order\_date` can be accurately converted from `DATE` to `TIMESTAMP\_NTZ`. Verify that any timestamps retrieved from this column behave as expected and maintain the correct precision.

#### 2. Nullability of New Fields:

- Confirm whether the newly added columns (`region\_code` and `order\_item\_id`) are nullable or required. If they are required, ensure existing applications or data insertions handle them appropriately.

#### 3. Referential Integrity:

- The addition of the `customers` table introduces potential foreign key relationships with the `customer\_id`. Verify that any existing data relationships remain intact and that data integrity is maintained in any new or modified queries.

#### 4. Integration Tests:

- Conduct integration tests with any applications or services that consume the `orders` and `order\_items` tables to ensure they correctly handle the modified data structure and any new relationships.

#### 5. Query Validation:

- Run existing queries that involve the `orders` and `order\_items` tables to validate that they still execute correctly and return expected results after the migration.

#### 6. Performance Testing:

- Assess any performance implications due to the addition of the `customers` table and changes in data types, especially if large volumes of data are expected.

By following these verification steps, potential migration risks can be addressed, ensuring a smooth transition from schema v1 to schema v2.

**Engineering counterpart**  
**3:12 AM - Responded in 0s**

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



## Agent response

3:12 AM - Responded in 11s

Certainly! Here's a detailed breakdown of my reasoning process, including the assumptions made at each step:

### ### 1. Identifying Breaking Changes

#### - Type Changes:

- I compared the data types of each column in the `orders` table between schema v1 and v2. The change from `DATE` to `TIMESTAMP\_NTZ` for the `order\_date` column is a significant alteration, as it affects how date and time data is stored and queried. My assumption was that changing a data type could lead to issues in applications that expect a specific data type.

#### - New Required Fields:

- I noted the addition of `region\_code` in the `orders` table and `order\_item\_id` in the `order\_items` table. I assumed that these fields could potentially be required based on usual schema evolution practices, where new identifiers or categorization fields often are. If they are indeed required (non-nullable), this would break any existing application logic that does not accommodate them.

#### - Removed Tables/Columns:

- I looked for any columns or tables that might have been removed in v2. Since none were found, my assumption was that the migration aimed to enhance the schema rather than strip away any existing functionality.

### ### 2. Identifying Non-Breaking Additions

#### - New Table:

- The addition of the `customers` table was noted as a non-breaking change, as it does not interfere with the existing tables or their relationships. I assumed that new data structures can generally be added without impacting existing functionality.

#### - New Columns:

- Similarly, the new columns in existing tables, such as `region\_code` and `order\_item\_id`, were identified as non-breaking. I assumed they would enhance functionality but not break existing queries or data handling if they were appropriately set up as nullable (which is common practice).

### ### 3. Assessing Migration Risks

#### - Data Type Conversion Risks:

- I assumed that converting a `DATE` to a `TIMESTAMP\_NTZ` could lead to compatibility issues in the application layer if there are assumptions about the data type or format. Thus, I emphasized the need for verification of data integrity post-migration.

#### - Nullability of New Fields:

- I assumed the new fields could be required based on common schema practices, which could break existing data entry or manipulation operations if they are not handled correctly.

#### - Referential Integrity:

- Adding a new table introduces potential foreign key relationships. I assumed that maintaining referential integrity between tables was crucial for data consistency, and any changes should be verified against existing relationships.

#### - Integration Tests:

- I assumed that existing applications or services that utilize the `orders` or `order\_items` tables would be impacted by these schema changes and would need to be tested to ensure continued functionality.

#### - Query Validation:

- I assumed that the logic in existing queries would need validation against the new schema to ensure they still operated correctly after the migration.

- Performance Testing:

- I considered that the addition of new tables or changing data types could have performance implications, especially if dealing with large datasets. Thus, I recommended performance testing to identify any potential issues early on.

### Summary of Assumptions

1. Data Type Changes Impact Applications: Any changes in data types can lead to compatibility issues.
2. New Fields May Be Required: New fields added to the schema may need to be handled as required in application logic.
3. Adding Tables is Non-Breaking: New tables generally do not affect existing functionality.
4. Data Integrity is Key: Referential integrity must be maintained during migration.
5. Existing Applications Need Testing: Existing applications must be validated against the new schema to ensure continued functionality.
6. Performance Implications Should Be Ass

**Engineering counterpart**  
**3:12 AM - Responded in 0s**

What would you verify before shipping this to production?

## Agent response

3:13 AM - Responded in 11s

Before shipping the schema changes from v1 to v2 to production, several verification steps should be carried out to ensure a smooth transition, data integrity, and application functionality. Here's a comprehensive list of what to verify:

### ### 1. Data Type Integrity Check

- Validate Data Conversion: Ensure that existing data in the `order\_date` column has been accurately migrated when changing from `DATE` to `TIMESTAMP\_NTZ`. Check for any data truncation or formatting issues.
- Type-Specific Queries: Run queries that specifically check the behavior of `TIMESTAMP\_NTZ` to confirm appropriate handling of time zones or date-time functions.

### ### 2. Nullability of New Fields

- Check Field Constraints: Identify whether `region\_code` in the `orders` table and `order\_item\_id` in the `order\_items` table are set as required (non-nullable). If they are, ensure that any data insertion or application logic correctly handles these fields.
- Data Population Tests: Conduct data insertion tests to verify that the new fields can accept data without errors and behave as expected when populated or left null.

### ### 3. Referential Integrity Verification

- Foreign Key Checks: If `customer\_id` in the `customers` table is supposed to reference the `customer\_id` in the `orders` table, check that the foreign key relationships are correctly implemented and enforced.
- Data Consistency: Ensure that there are no orphaned records (e.g., orders without corresponding customers) after migration.

### ### 4. Integration Testing

- End-to-End Tests: Run integration tests on all services or applications that interact with the `orders`, `order\_items`, and new `customers` tables. Validate that they perform as expected and return correct results with the modified schema.
- Regression Testing: Conduct regression testing on existing functionalities to ensure that they have not been negatively impacted by the schema changes.

### ### 5. Query Validation

- Test Existing Queries: Execute existing queries that utilize the modified tables to ensure results remain consistent with previous outputs. Check for optimization issues or changes in execution plans.
- New Query Scenarios: Create and run new queries that take advantage of the new fields and table additions to verify they function as intended.

### ### 6. Performance Testing

- Benchmark Queries: Measure performance benchmarks for relevant queries before and after the schema changes to evaluate any performance degradation or improvements.
- Load Testing: Conduct load testing to simulate high-volume transactions and assess how the new schema performs under stress.

### ### 7. Backup and Rollback Plan

- Backup Current Data: Ensure that backups of the current database schema and data are taken before applying changes.
- Rollback Strategy: Develop a clear rollback strategy in case the migration introduces critical failures. This should include scripts and procedures to revert to the previous schema and data state.

### ### 8. Documentation Updates

- Schema Documentation: Update any relevant documentation to reflect changes in the schema, including new columns, data types, and any new tables.
- API Changes: If applicable, update any API documentation to account for additional fields, new endpoints, or modified return types.

### ### 9. User Acceptance Testing (UAT)

- Stakeholder Review: If possible, conduct user acceptance testing with key stakeholders to ensure that the new schema meets business requirements and expectations.
- Feedback Mechanism: Set up a mechanism for gathering feedback from users interacting with the new schema post-deployment.

#### ### Final Steps

After completing the above verifications and ensuring that all aspects are functioning as expected, one should consider deploying the changes during a low-traffic period to minimize the impact on users. Monitor the system closely after deployment for any issues that may arise.

**Engineering counterpart**  
**3:13 AM - Responded in 0s**

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

## Agent response

3:13 AM - Responded in 11s

### ### Top Risks Associated with the Schema Migration

#### 1. Data Type Conversion Issues:

- Changing ``order_date`` from ``DATE`` to ``TIMESTAMP_NTZ`` could lead to data integrity issues if existing date values are not compatible with the new type. There could be unexpected behavior with date manipulations or queries.

#### 2. Nullability Constraints:

- If new fields such as ``region_code`` or ``order_item_id`` are not properly set up to handle null values, this could lead to errors when inserting or updating records, affecting application functionality.

#### 3. Referential Integrity Violations:

- The introduction of the ``customers`` table and any related foreign key constraints could lead to violations if existing data does not align correctly. For example, if there are orders referencing non-existent customers.

#### 4. Application Breakage:

- Existing applications or services that depend on the schemas could break if they are not updated to accommodate the new fields or data types. This could result in runtime errors or incorrect data processing.

#### 5. Performance Degradation:

- The addition of new fields or changes in data types may cause performance issues in existing queries due to changes in execution plans or increased resource consumption.

#### 6. Data Migration Failures:

- If the migration process does not handle existing data correctly (e.g., missing data in new columns or incorrect data types), this could lead to data loss or corruption.

### ### Rollback Plan

In the event that the migration does not work as intended, having a robust rollback plan is essential to restore the system to its previous state. Here is a step-by-step rollback plan:

#### 1. Backup Current Data:

- Prior to executing any migration scripts, take a complete backup of the existing database, including the current schema and all data. This ensures that you can restore the system to its exact state before the migration.

#### 2. Document Migration State:

- Keep detailed logs of the changes made during the migration, including timestamps and the exact schema alterations. This will help identify what needs to be reversed.

#### 3. Rollback Scripts:

- Prepare and test rollback scripts ahead of time to revert the schema changes. These scripts should include:
  - Dropping any newly added tables (like the ``customers`` table).
  - Altering modified tables back to their original structure, including any necessary data type changes (e.g., reverting ``order_date`` from ``TIMESTAMP_NTZ`` back to ``DATE``).
  - Removing any new columns added to existing tables (like ``region_code`` and ``order_item_id``).

#### 4. Data Restoration:

- Use the backup taken prior to the migration to restore the data. This may involve restoring the whole database or selectively restoring tables affected by the migration.

#### 5. Validation After Rollback:

- Once the rollback is executed, thoroughly validate that the database is back to its original state. This includes:
  - Ensuring that all data is intact and consistent.

- Running any existing queries to confirm they work as before.
- Checking application functionality to verify that it operates correctly with the original schema.

#### 6. Communication and Monitoring:

- If rollback is necessary, communicate with stakeholders about the issues encountered and the steps taken to mitigate them.
- After rollback, monitor the system for any lingering issues that may arise from the migration attempt.

#### 7. Post-Mortem Analysis:

- After resolving the situation, conduct a post-mortem analysis to identify the root cause of the issues encountered during migration. This should inform future migration strategies to avoid similar pitfalls.

By preparing for these risks and having a clear rollback plan in place, the migration process can proceed more confidently, with contingency strategies ready to be enacted if needed.



**Engineering counterpart**

**3:13 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.