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: deepseek/deepseek-chat
Roles: Data Engineer (AI agent) <> Engineering counterpart (AI agent)
Duration: 1m 21s
Actual run cost: $0.002985
Projected execution cost: $1.97/mo (30/biz day × 22 days)
Report date: July 13, 2026 - 11:08 PM EST

Session: sample-lb-rankings-seed-069ec178-232b-4665-a7ad-92f81b2bd752

de_schema_drift_v1 - agent - openrouter - deepseek/deepseek-chat - ended

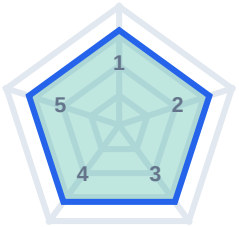
Overall

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: 11268 chars - Latency: 81s - 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 21s

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
3:07 AM - Responded in 20s

Schema Migration Review: v1 (2024-Q1) vs. v2 (2024-Q2)

---

1. Breaking Changes
These changes could cause issues with existing applications or queries:

- Data Type Change:
  - Table: `orders`
  - Column: `order_date`
  - Change: `DATE` -> `TIMESTAMP_NTZ`
    - Applications relying on `DATE` formatting or behavior may break.
    - Data truncation or precision issues could occur if `TIMESTAMP_NTZ` includes time components.

- New Required Column:
  - Table: `order_items`
  - Column: `order_item_id`
    - This column is added as a primary key or unique identifier. Existing data must be migrated with valid values for this column.

- New Table:
  - Table: `customers`
    - This table is added, which is not inherently breaking but may impact applications expecting only `orders` and `order_items` tables.

---

2. Non-Breaking Additions
These changes are additive and unlikely to break existing applications:

- New Column:
  - Table: `orders`
  - Column: `region_code`
    - This is an optional addition (`VARCHAR(8)`), so existing queries will not break.

---

3. Migration Risks and Verification Steps
Risks:
1. Data Type Migration:
    - Converting `order_date` from `DATE` to `TIMESTAMP_NTZ` may introduce unintended time components (e.g., defaulting to `00:00:00`).
    - Verify that historical `order_date` data retains the correct semantics after migration.

2. New Required Column:
    - Adding `order_item_id` to `order_items` requires generating unique values for existing rows.
    - Ensure uniqueness and consistency in the migration process.

3. New Table:
    - The `customers` table introduces new relationships. Ensure that referential integrity is maintained with `orders.customer_id`.

4. Query Compatibility:

- Applications querying `order_date` as a `DATE` may fail due to the new `TIMESTAMP_NTZ` type.
- Applications expecting only `orders` and `order_items` tables may need updates to handle the new `customers` table.

Verification Steps:
1. Validate that `order_date` values in `orders` are correctly migrated to `TIMESTAMP_NTZ` format.
2. Ensure `order_item_id` in `order_items` is populated with unique and non-null values.
3. Confirm that all `orders.customer_id` values have corresponding entries in `customers.customer_id`.
4. Test existing application queries and ETL pipelines to ensure compatibility with the new schema.
5. Verify that default values or application logic handles the new `region_code` column appropriately.
6. Perform regression testing to ensure no downstream systems are affected by the schema changes.

By addressing these changes and risks proactively, the migration can be executed smoothly with minimal disruption.

Engineering counterpart
3:07 AM - Responded in 0s

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

Agent response
3:08 AM - Responded in 29s

Step-by-Step Reasoning for Schema Migration Review

Objective: Identify breaking changes, non-breaking additions, and migration risks between v1 (2024-Q1) and v2 (2024-Q2) schemas.

Step 1: Compare Tables and Columns

1.1 Tables in v1 (2024-Q1)
- `orders`
  - `order_id` (NUMBER)
  - `customer_id` (NUMBER)
  - `order_date` (DATE)
  - `total_usd` (NUMBER(12,2))
  - `status` (VARCHAR(32))
- `order_items`
  - `order_id` (NUMBER)
  - `sku` (VARCHAR(64))
  - `qty` (NUMBER)
  - `unit_price` (NUMBER(10,2))

1.2 Tables in v2 (2024-Q2)
- `orders`
  - `order_id` (NUMBER)
  - `customer_id` (NUMBER)
  - `order_date` (TIMESTAMP_NTZ) (Changed type)
  - `total_usd` (NUMBER(12,2)) (Unchanged)
  - `status` (VARCHAR(32)) (Unchanged)
  - `region_code` (VARCHAR(8)) (New column)
- `order_items`
  - `order_item_id` (NUMBER) (New column)
  - `order_id` (NUMBER) (Unchanged)
  - `sku` (VARCHAR(64)) (Unchanged)
  - `qty` (NUMBER) (Unchanged)
  - `unit_price` (NUMBER(10,2)) (Unchanged)
- `customers` (New table)
  - `customer_id` (NUMBER)
  - `region` (VARCHAR(32))
  - `segment` (VARCHAR(16))

Step 2: Identify Breaking Changes

2.1 Data Type Change (`orders.order_date`)
- Change: `DATE` -> `TIMESTAMP_NTZ`
- Why Breaking?
  - Applications or queries expecting a `DATE` may fail if they rely on `DATE`-specific functions (e.g., `DATE_TRUNC`).
  - A `TIMESTAMP` includes time components (`YYYY-MM-DD HH:MM:SS`), while a `DATE` is just `YYYY-MM-DD`.
  - If existing queries filter by date without time handling, they may return incorrect results.

2.2 New Required Column (`order_items.order_item_id`)

- Change: Added `order_item_id` (NUMBER)
- Why Breaking?
  - If `order_item_id` is meant to be a primary key, existing data must be backfilled with unique values.
  - Any `INSERT` statements that don't specify `order_item_id` will fail unless it has a default value.

2.3 New Table (`customers`)
- Not inherently breaking, but can cause issues if:
  - There are foreign key constraints linking `orders.customer_id` to `customers.customer_id`.
  - Applications assume only two tables (`orders`, `order_items`) exist.

Step 3: Identify Non-Breaking Additions

3.1 New Optional Column (`orders.region_code`)
- Change: Added `region_code` (VARCHAR(8))
- Why Non-Breaking?

Engineering counterpart
3:08 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response
3:08 AM - Responded in 14s

Before shipping this schema migration to production, I would verify the following key areas to ensure a smooth transition and minimize the risk of breaking existing systems:

---

1. Data Type Migration (`orders.order_date`)
- Verification Steps:
  1. Confirm that all existing `order_date` values (from `DATE`) are correctly migrated to `TIMESTAMP_NTZ`.
     - Ensure no unintended time components (e.g., `00:00:00`) are added during migration.
  2. Test queries that filter or manipulate `order_date` to ensure they still work as expected with `TIMESTAMP_NTZ`.
  3. Validate that downstream systems (e.g., reporting, analytics) can handle the new `TIMESTAMP_NTZ` format.
  4. Check for any implicit assumptions in application code (e.g., date truncation, formatting logic).

---

2. New Required Column (`order_items.order_item_id`)
- Verification Steps:
  1. Ensure `order_item_id` is populated with unique, non-null values for all existing rows in `order_items`.
  2. Validate that `order_item_id` is correctly indexed or designated as a primary key.
  3. Test `INSERT` statements to confirm they work with the new column.
     - If `order_item_id` has no default value, ensure applications are updated to provide it.
  4. Check for any downstream systems (e.g., ETL pipelines, APIs) that interact with `order_items` to ensure they handle the new column.

---

3. New Table (`customers`) and Relationships
- Verification Steps:
  1. Confirm that all `customer_id` values in `orders` have corresponding entries in `customers.customer_id`.
  2. Verify that foreign key constraints (if any) are correctly enforced.
  3. Test queries that join `orders` and `customers` to ensure they return expected results.
  4. Validate that applications referencing `customers` are updated to handle the new table.

---

4. New Optional Column (`orders.region_code`)
- Verification Steps:
  1. Confirm that `region_code` is nullable or has a default value for existing rows.
  2. Test queries that include or exclude `region_code` to ensure they behave as expected.
  3. Validate that downstream systems (e.g., reporting, analytics) can handle the new column.

---

5. Application Compatibility
- Verification Steps:
  1. Perform regression testing on all applications that interact with the database to ensure they work with the new schema.

  2. Test ETL pipelines, APIs, and scheduled jobs to confirm they handle the changes correctly.
  3. Verify that existing queries (e.g., SELECT, INSERT, UPDATE) are compatible with the new schema.

---

6. Data Integrity

- Verification Steps:
  1. Run data validation checks to ensure no data loss or corruption occurred during migration.
  2. Confirm that indexes, constraints, and relationships are preserved and functioning correctly.
  3. Validate that all primary keys are unique and non-null.

---

7. Rollback Plan
- Verification Steps:
  1. Ensure a rollback plan is in place in case migration issues arise.
  2. Test the rollback process to confirm it restores the database to its pre-migration state.

---

8. Performance Testing
- Verification Steps:
  1. Test the performance of queries involving the new columns and tables to ensure they meet acceptable thresholds.
  2. Validate that indexes and

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 18s

Top Risks in This Schema Migration

1. Data Corruption or Loss
- Risk: If the `order_date` migration from `DATE` to `TIMESTAMP_NTZ` fails, timestamps could be incorrect or truncated.
- Mitigation: Back up the `orders` table before migration and validate data post-migration.

2. Application Breakage Due to Type Changes
- Risk: Applications expecting `DATE` may fail when reading `TIMESTAMP_NTZ`.
- Mitigation: Test all date-handling logic in applications before rollout.

3. Missing `order_item_id` Values
- Risk: If `order_item_id` is required but not backfilled, inserts will fail.
- Mitigation: Ensure default values (e.g., auto-increment) are set or backfill existing rows.

4. Referential Integrity Issues
- Risk: If `customers` table is missing records for `orders.customer_id`, joins will break.
- Mitigation: Pre-migration, validate that all `customer_id` values in `orders` exist in `customers`.

5. Performance Degradation
- Risk: New columns (`region_code`) and indexes (`order_item_id`) may slow down queries.
- Mitigation: Benchmark query performance before and after migration.

Rollback Plan (If Migration Fails)

1. Immediate Rollback Steps
1. Stop all writes to the affected tables (`orders`, `order_items`, `customers`).
2. Restore from backup (if available) to revert to the pre-migration schema.
3. If backup isn't feasible, reverse the changes manually:
   - Drop `region_code` from `orders`.
   - Drop `order_item_id` from `order_items` (if it was added).
   - Revert `order_date` from `TIMESTAMP_NTZ` back to `DATE`.
   - Drop the `customers` table if it was newly created.

2. Post-Rollback Verification
- Data Integrity Check: Ensure all data matches the pre-migration state.
- Application Smoke Test: Confirm apps work as expected with the old schema.
- Root Cause Analysis: Document why the migration failed and fix before retrying.

3. Fallback Strategy
- If a full rollback isn't possible, deploy a hotfix to make applications compatible with the new schema temporarily.

Key Takeaways
? Test thoroughly in staging before production.
? Back up data before migration.
? Monitor closely after deployment.
? Have a rollback playbook ready.

Would you like a detailed step-by-step rollback SQL script for this scenario?

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.