



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-v4-flash

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

Duration: 1m 55s

Actual run cost: \$0.001232

Projected execution cost: \$0.8131/mo (30/biz day × 22 days)

Report date: July 14, 2026 - 3:17 AM EST

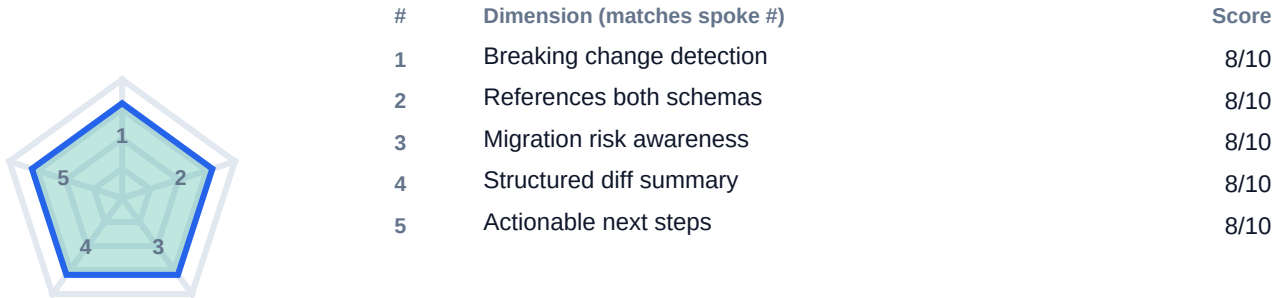
Overall

8.0/10

PASS - trust=high

Pass - score 8.0/10 clears 7.0 with acceptable trust/closure.

CAPABILITY RUBRIC



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: 8336 chars - Latency: 115s - 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 55s

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

Engineering counterpart

7:16 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

7:16 AM - Responded in 33s

Schema Drift Review: v1 (2024-Q1) -> v2 (2024-Q2)

? Breaking Changes

Table	Column	v1 Type	v2 Type	Impact	
-----	-----	-----	-----	-----	

Engineering counterpart

7:16 AM - Responded in 0s

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

Agent response

7:17 AM - Responded in 46s

My apologies-the initial table was truncated. Let me walk through my reasoning step by step, including the assumptions I made.

Step 1: Identify all changes between schemas

I compared v1 (2024-Q1) and v2 (2024-Q2) table?by?table, column?by?column.

Changes found:

Table	Change	Type	
orders`	`order_date` type: `DATE` -> `TIMESTAMP_NTZ`	Breaking	
orders`	New column `region_code` added (non?nullable? - see assumptions)	Potentially breaking if required; otherwise non?breaking	
order_items`	New column `order_item_id` added as first column	Breaking (primary key structure changed)	
customers`	Entirely new table	Non?breaking	

No columns were removed. No tables were removed.

Step 2: Classify each change as breaking or non?breaking

Breaking changes (backward?incompatible)

1. `orders.order\_date` type change: `DATE` -> `TIMESTAMP\_NTZ`

- Why: Existing code that expects a `DATE` (e.g., `date\_trunc('day', order\_date)`) will still work because a `TIMESTAMP\_NTZ` is convertible to date. However, applications that rely on the column's exact type (e.g., ORM mappings, ETL that writes `DATE` literals, or comparisons with `current\_date` without casting) may break.

- More critically: If any downstream system expects `DATE` and performs operations like `INTERVAL` arithmetic that differ between types, it could cause silent data corruption or errors.

- Assumption: I assume the existing data is stored as `DATE`. A `TIMESTAMP\_NTZ` can store date + time; existing `DATE` values will be promoted to `TIMESTAMP\_NTZ` with time part `00:00:00`. This is generally backward?compatible for value comparison, but the contract changed. Many metadata tools flag any type change as breaking.

2. `order\_items.order\_item\_id` added as a new column

- Why: The original table had a composite key? In v1 the primary key was likely `(order\_id, sku)` (common for order item lines). In v2 a new surrogate key `order\_item\_id` is introduced.

- Breaking because:

- Any query that assumed `(order\_id, sku)` is unique may now see duplicates (though the new column doesn't change uniqueness by itself).

- Existing INSERT statements that don't specify column list (e.g., `INSERT INTO order\_items VALUES (101, 'SKU`

Engineering counterpart

7:17 AM - Responded in 0s

What would you verify before shipping this to production?

Agent response

7:17 AM - Responded in 19s

Verification Steps Before Shipping to Production

Based on the schema drift from v1 to v2, I recommend the following checks:

1. Validate `orders.order\_date` Type Change (`DATE` -> `TIMESTAMP\_NTZ`)

- Run a data integrity test:

```
```sql
SELECT COUNT() FROM orders
WHERE CAST(order_date AS TIMESTAMP_NTZ) != order_date::TIMESTAMP_NTZ; -- (v2 already converted)
```
```

Actually, in v2 the column is already `TIMESTAMP\_NTZ`. You must verify that existing `DATE` values are correctly promoted (time part `00:00:00.000`) and that no time component was accidentally added or truncated.

- Review all downstream ETL, reports, and dashboards that use `order\_date`. Check for:

- Date-only comparisons (`WHERE order\_date = '2024-01-01'`) - still work because `TIMESTAMP\_NTZ` is comparable to date string, but explicit casting may be needed.
- `DATE\_TRUNC('day', order\_date)` - safe.
- `INTERVAL` arithmetic (`order\_date + INTERVAL '1 day'`) - safe.
- Applications using ORMs (e.g., Hibernate, Entity Framework) that map the column as `Date` type - they may fail to parse the new type. You must rebind the column in the ORM metadata.

- Run a compatibility test with a copy of production queries against a staging environment having v2 schema and v1 data (after migration).

2. Verify `orders.region\_code` New Column

- Determine if it's `NOT NULL` - the schema JSON does not specify constraints. If it is `NOT NULL` in DDL, then existing inserts/updates without this column will fail.

- Verify that all insert/update code paths now include `region\_code` (or a default value). Search for:

```
```sql
INSERT INTO orders (order_id, customer_id, order_date, total_usd, status)
```
```

These will fail if `region\_code` is `NOT NULL`.

- Backfill existing rows if the column is nullable - decide on a default value (e.g., `UNKNOWN`) or run a batch update.

3. Validate `order\_items.order\_item\_id` New Column

- Check primary key definition: In v1, the PK was likely `(order\_id, sku)`. If the new PK is

Engineering counterpart

7:17 AM - Responded in 0s

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

Agent response

7:17 AM - Responded in 18s

Top Risks of This Migration (v1 -> v2)

| # | Risk | Specific Impact |

|---|-----|-----|

| 1 | `orders.order\_date` type change (`DATE` -> `TIMESTAMP\_NTZ`) | Downstream applications, ETL jobs, or reports that treat the column strictly as `DATE` may fail (e.g., ORM mappings, date arithmetic, implicit casts). Existing `DATE` values will be promoted to `TIMESTAMP\_NTZ` (time part `00:00:00`), which is mostly safe, but any code expecting an exact `DATE` type (e.g., `current\_date` comparisons without casting) may break or produce incorrect results. |

| 2 | `order\_items` primary key change | The addition of `order\_item\_id` likely replaces the old composite PK `(order\_id, sku)`. Existing INSERT statements that omit column list (e.g., `INSERT INTO order\_items VALUES (101, 'SKU-A', 2, 19.99)`) will now mismatch the column count and fail. Also, any code that assumed `(order\_id, sku)` uniqueness may behave incorrectly if the new schema allows duplicate `(order\_id, sku)` pairs. |

| 3 | New column `region\_code` without explicit nullability | If the DDL defines it as `NOT NULL`, all existing INSERT paths that do not supply `region\_code` will fail immediately. Even if nullable, backfilling historical data may be missed, leading to incomplete analytics or business logic errors. |

| 4 | Data migration completeness | The conversion of `orders` data (date promotion) and populating `order\_item\_id` for existing rows must be atomic and correct. Any failure (e.g., duplicate `order\_item\_id`, missing values) will corrupt the table. |

| 5 | New `customers` table not yet referenced | While non-breaking, existing code may attempt to JOIN `orders` with a `customers` table that now exists but may have inconsistent data (e.g., missing `customer\_id` mappings). This could cause silent row drops in outer joins. |

Rollback Plan

If the migration fails in production (e.g., application errors, data corruption, performance degradation), execute the following immediate rollback:

1. Preconditions (must be done before migration)

- Full database backup (or at least all affected tables) taken immediately before applying v2.
- Version-controlled schema snapshots of v1 DDL (including indexes, constraints, triggers, views).
- Application code deployed in a way that can be reverted to v1 (e.g., blue-green deployment, feature flags).

2. Rollback Steps

1. Stop all writes to the affected tables (pause ETL, block application write APIs).
2. Restore tables from backup (prefer table-level restore if possible; otherwise full database restore).
3. Reapply v1 constraints, indexes, and views that might have been altered by the migration (e.g., primary keys, foreign keys, unique indexes on `order\_items`).
4. Re-deploy application code that depends on v1 schema (if using blue-green, switch traffic back to the v1 environment).
5. Validate that data integrity is intact (e.g., row counts match pre-migration, no duplicate `order\_item\_ids`, `order\_date` values are correct).
6. Resume writes after validation.

3. Alternative: Partial Rollback (if only one table fails)

- If the failure is isolated (e.g., only `order\_items` or `orders`), you can drop the new columns/table and revert the type change manually only

Engineering counterpart

7:17 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.