Fraud Claims Detection - Data Observability

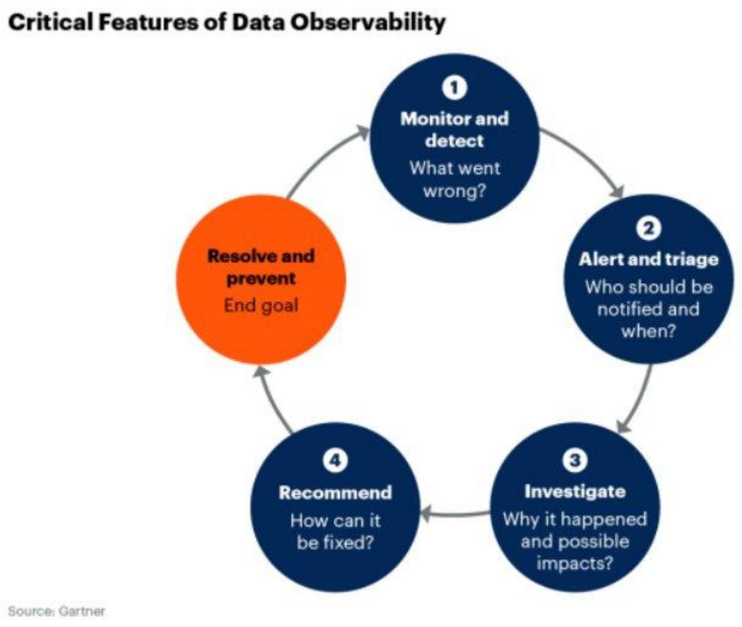
Introduction

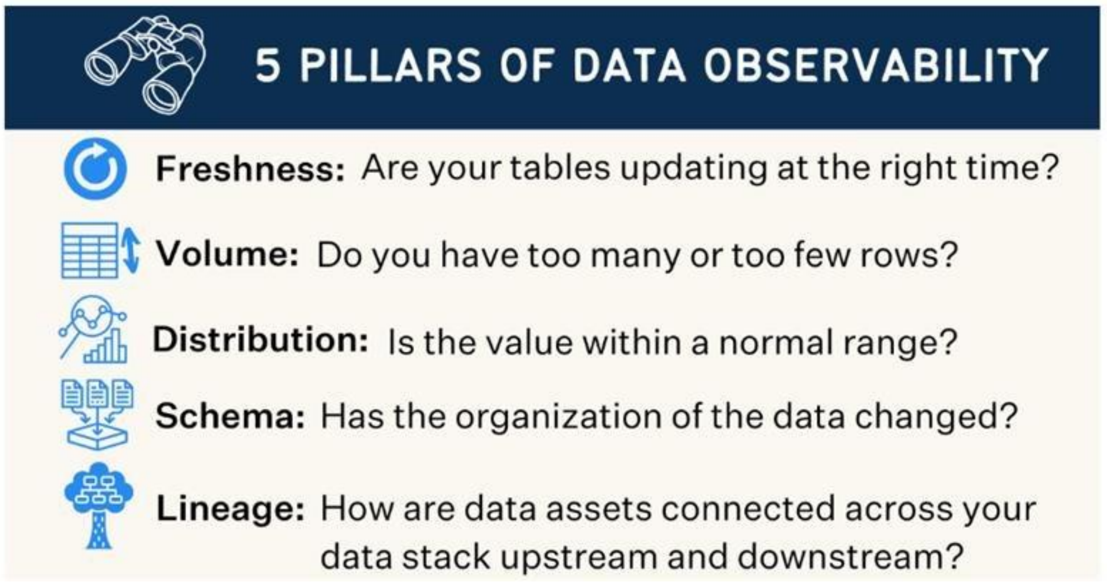
Data quality and observability are essential to ensure that the data used across analytics, reporting, and business processes is reliable and trustworthy. As data moves through different layers in the platform, issues such as missing values, schema changes, delays in updates, or unexpected volume variations can impact downstream outputs.

This document outlines the Data Quality and Data Observability framework implemented in our project. It covers how we monitor data freshness, volume, schema changes, and key quality rules, along with dashboards and alerting mechanisms used for proactive issue detection. The goal is to provide a clear, standardized approach to maintaining consistent and accurate data across the platform.

Data Observability

Data Observability provides end-to-end visibility into the health of data pipelines. It helps monitor whether data is arriving on time (freshness), whether the record counts behave as expected (volume), and whether any unexpected structural changes occur (schema). By continuously tracking these patterns, Data Observability enables early detection of anomalies, reduces troubleshooting time, and ensures consistent and reliable data delivery across all layers of the platform.




5 PILLARS OF DATA OBSERVABILITY

Freshness: Are your tables updating at the right time?

Volume: Do you have too many or too few rows?

Distribution: Is the value within a normal range?

Schema: Has the organization of the data changed?

Lineage: How are data assets connected across your data stack upstream and downstream?

Implementation Overview

The Data Observability implementation begins in Microsoft Fabric using a single notebook that performs all the required checks. The notebook is responsible for creating the observability tables, processing the pipeline metadata, and running the logic for freshness, volume change, schema change, and data quality validations. All outputs from this notebook are stored inside the Lakehouse and later used to build the Power BI observability dashboard.

Below is the flow used in the implementation:

1. **Created Observability Tables in the Lakehouse**
   Using PySpark, the notebook creates the required tables for storing:
    a. Freshness results
    b. Volume change results
    c. Schema change results
    d. Data quality results
2. **Implemented Observability Logic in the Notebook**
   The notebook reads the metadata for each table and applies the observability checks.
   Each check produces a structured output that is written back to the Lakehouse.
3. **Connected Power BI to These Tables**
   The Power BI dashboard directly uses these Lakehouse tables as its data source, allowing automatic visualization of freshness status, volume trends, schema change flags, and DQ test results.

Freshness Check

Objective

To ensure that data in each table is up-to-date according to the expected refresh interval. Any delays beyond the defined threshold are flagged as Critical.

Implementation Logic

1. Load the table metadata or Delta table.
2. Calculate the latest timestamp in the table.
3. Compare it with the current timestamp to compute delay in days.
4. Determine the threshold dynamically based on the layer:
    a. `raw`, `curate`, `publish`: 180 days
    b. `eventhouse`: 7 days
    c. Others: 30 days (default)
5. Assign status based on the delay:
    a. `Fresh` if delay ≤ threshold
    b. `Critical` if delay > threshold
6. Append results to the freshness results table.

```
from datetime import datetime
from pyspark.sql.functions import to_timestamp, col

df = spark.table(cfg["table"])  # Or load Delta path for eventhouse
df = df.withColumn("ts", to_timestamp(col(cfg["timestamp_col"])))
latest_ts = df.agg({"ts": "max"}).collect()[0][0]
now_ts = datetime.utcnow()
delay_days = int((now_ts - latest_ts).total_seconds() / (60 * 60 * 24))

threshold_days = 180 if cfg["layer"] in ["raw", "curate", "publish"] else 7
status = "Fresh" if delay_days <= threshold_days else "Critical"
```

Schema Change Check

Objective

To monitor structural changes in all Lakehouse and Eventhouse tables. This includes detecting:

- Added columns
- Removed columns
- Any schema differences across runs

The output is stored in a schema change log table for audit and dashboard visualization.

Implementation Logic

1. Prepare a list of tables - All tables from raw, curate, publish, and eventhouse are included.
2. Load each table
    a. Normal tables are loaded using `spark.table()`.
    b. Eventhouse tables are loaded using their Delta file path.
3. Read the current schema - Convert the table's schema into a simple JSON format.
4. Get the previous schema - Fetch the last saved schema record from the SchemaChangeLog table.
5. Compare both schemas - Identify if any columns were added or removed.
6. Decide the status
    a. If there is any difference → Changed
    b. If nothing changed → No Change

7. Store the result - Save the table name, schema, status, and timestamp into the SchemaChangeLog table.

```python
# Extract current schema as JSON
current_schema_json = get_schema_json(df)

# Get last captured schema
prev_schema_json = prev_row[0]["Schema"] if prev_row else None

# Compare column names
current_fields = set(f['name'] for f in json.loads(current_schema_json))

# Handle previous schema separately for easy screenshot
if prev_schema_json:
    previous_fields = set(f['name'] for f in json.loads(prev_schema_json))
else:
    previous_fields = set()

# Identify added and removed columns
added = list(current_fields - previous_fields)
removed = list(previous_fields - current_fields)

# Set status
status = "Changed" if added or removed else "No Change"
diff = f"Added: {added}, Removed: {removed}" if status == "Changed" else "No Change"
```

Volume Check

Objective

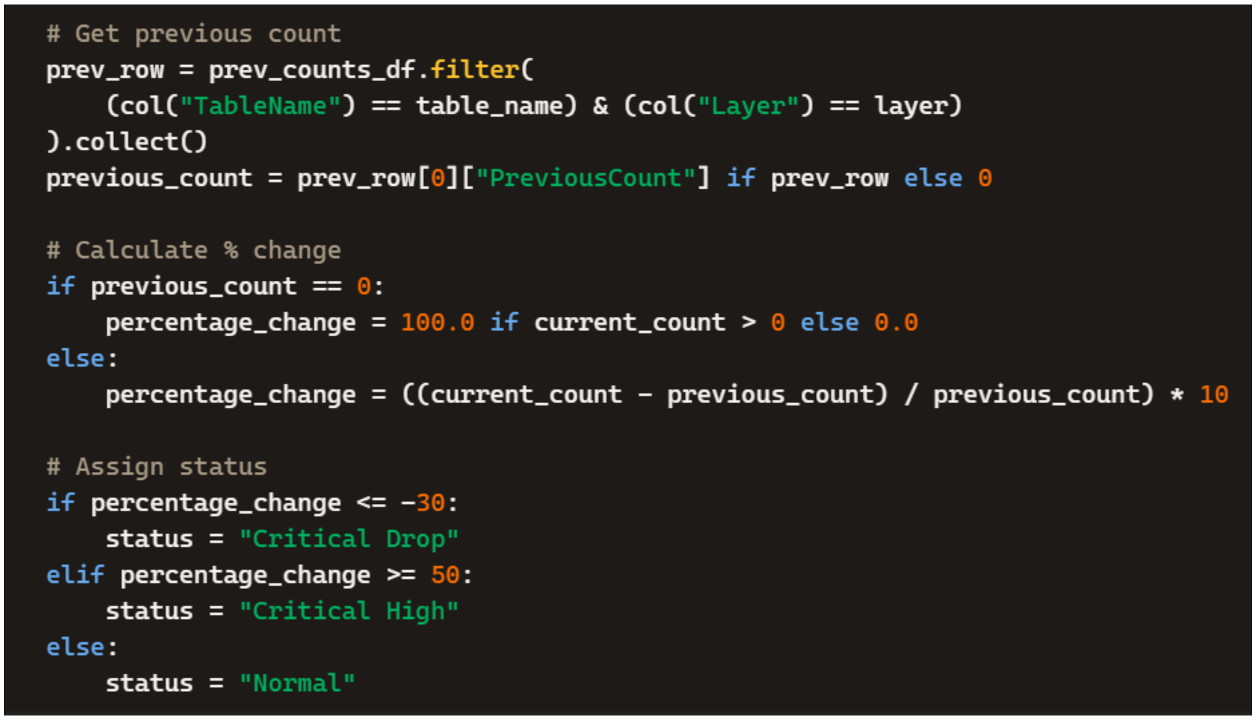
Volume Check tracks how many rows each table has in every run.
It helps identify:

- If the table volume increased
- If the volume dropped unexpectedly
- If the volume stayed the same

This is useful for detecting missing data, partial loads, or abnormal growth.

Implementation Logic

1. Start monitoring and log the timestamp.
2. Define the list of tables/paths to check (raw, curate, publish, eventhouse).
3. For each table:
    a. Load the table (different method for eventhouse paths).
    b. Extract the current schema (column names + types).
    c. Compare with the previous schema stored in `SchemaChangeLog`
    d. Identify added or removed columns.
    e. Assign status:
        i. Changed → if columns were added/removed
        ii. No Change → if schema is the same
    f. Record the difference
4. Save results into `raw.dataquality.SchemaChangeLog` for dashboards/alerts.



```
# Get previous count
prev_row = prev_counts_df.filter(
    (col("TableName") == table_name) & (col("Layer") == layer)
).collect()
previous_count = prev_row[0]["PreviousCount"] if prev_row else 0

# Calculate % change
if previous_count == 0:
    percentage_change = 100.0 if current_count > 0 else 0.0
else:
    percentage_change = ((current_count - previous_count) / previous_count) * 10

# Assign status
if percentage_change <= -30:
    status = "Critical Drop"
elif percentage_change >= 50:
    status = "Critical High"
else:
    status = "Normal"
```

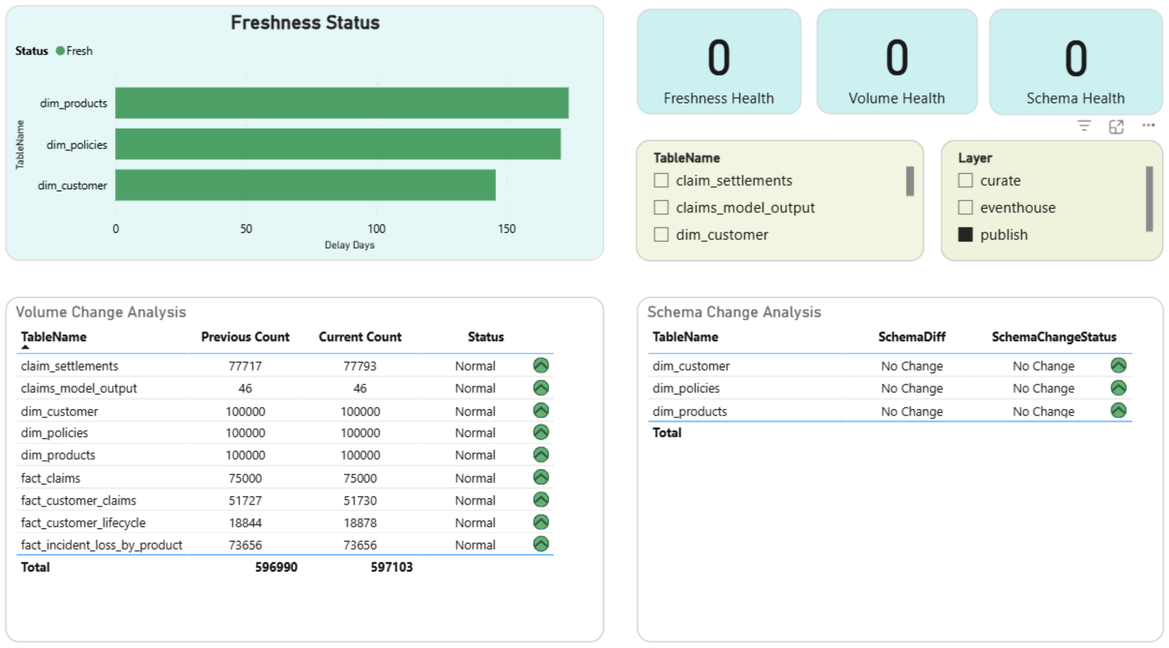
Data Observability Dashboard

This dashboard is created using the observability result tables generated from the notebook.

It provide a quick view of:

- Freshness status
- Volume changes
- Schema change detection

It helps monitor the overall health of all pipelines at a glance.



Volume Change Analysis

TableName	Previous Count	Current Count	Status	
claim_settlements	77717	77793	Normal	🟢
claims_model_output	46	46	Normal	🟢
dim_customer	100000	100000	Normal	🟢
dim_policies	100000	100000	Normal	🟢
dim_products	100000	100000	Normal	🟢
fact_claims	75000	75000	Normal	🟢
fact_customer_claims	51727	51730	Normal	🟢
fact_customer_lifecycle	18844	18878	Normal	🟢
fact_incident_loss_by_product	73656	73656	Normal	🟢
Total	**596990**	**597103**		

Schema Change Analysis

TableName	SchemaDiff	SchemaChangeStatus	
dim_customer	No Change	No Change	🟢
dim_policies	No Change	No Change	🟢
dim_products	No Change	No Change	🟢
Total			

Data Quality

Data Quality checks make sure the data coming into the pipeline is clean, complete, and correct.
Your code checks things like:

- Null checks → make sure important fields are not empty
- Data type format checks → e.g., date must look like a date
- Value checks → certain fields must match allowed values
- Duplicate checks → catch repeated entries

These checks help you avoid bad data entering downstream systems.

Data quality tables created via notebooks for dashboard consumption.

- `DataQualityMaster` records each run of the framework, storing the run ID, start and end times, trigger mode (manual or scheduled), status, and remarks. It acts as the central log of all runs.
- `DataQualityTestCases` contains the catalog of active test cases, where each row defines a rule with its SQL logic, the source table, and layer. This lets you manage rules dynamically without changing code.
- `DataQualityExecutions` captures the detailed results of each test case during a run, including how many rows were scanned, how many failed, the pass rate, status, and execution time.
- `DataQualityExecutionErrors` stores error details whenever a test case fails or produces violations, including sampled error rows or exception messages, linked back to the run and test case.
- `DQRunSummary` provides a high-level summary of the entire run, showing total test cases, how many passed or failed, duration, overall status, and the list of test case IDs.

Together, these tables form a complete monitoring system: the **Master** table registers the run, **TestCases** defines the rules, **Executions** logs the results, **ExecutionErrors** captures problems, and **RunSummary** gives the overall picture.

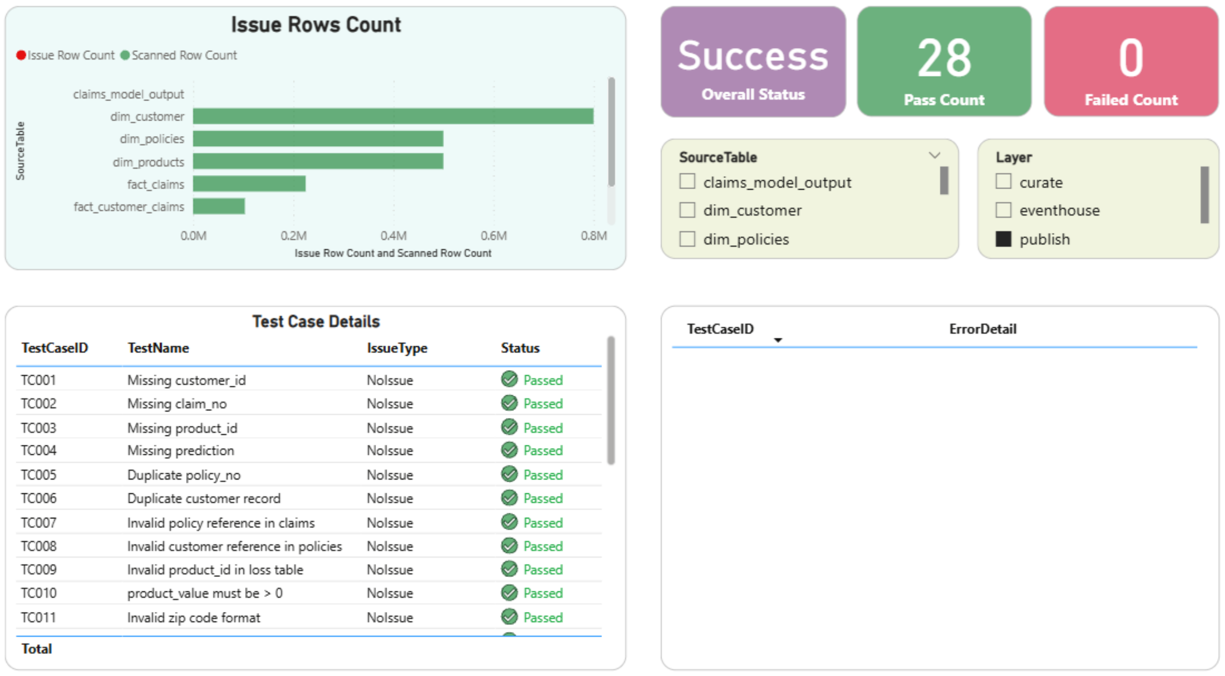
Data Quality Dashboard

The dashboard is built using the data quality result tables produced by the data quality notebook.

It shows:

- Null/Completeness checks
- Invalid/failed rule checks
- Duplicate/format validations

It helps identify and track data quality issues across datasets.



Success
Overall Status

28
Pass Count

0
Failed Count

SourceTable
- ☐ claims_model_output
- ☐ dim_customer
- ☐ dim_policies

Layer
- ☐ curate
- ☐ eventhouse
- ■ publish

Test Case Details

TestCaseID	TestName	IssueType	Status
TC001	Missing customer_id	NoIssue	✓ Passed
TC002	Missing claim_no	NoIssue	✓ Passed
TC003	Missing product_id	NoIssue	✓ Passed
TC004	Missing prediction	NoIssue	✓ Passed
TC005	Duplicate policy_no	NoIssue	✓ Passed
TC006	Duplicate customer record	NoIssue	✓ Passed
TC007	Invalid policy reference in claims	NoIssue	✓ Passed
TC008	Invalid customer reference in policies	NoIssue	✓ Passed
TC009	Invalid product_id in loss table	NoIssue	✓ Passed
TC010	product_value must be > 0	NoIssue	✓ Passed
TC011	Invalid zip code format	NoIssue	✓ Passed
Total			

TestCaseID	ErrorDetail

Real-Time Observability Patterns

The real-time dashboard was created using KQL queries on the Eventhouse monitoring tables. These queries help track system and data behaviour instantly, without waiting for scheduled notebook runs. Below are the key real-time observability patterns included in the dashboard.

1. Freshness Status (Real-Time Ingestion Monitoring)

This pattern monitors the most recent ingestion time for each Eventhouse table. It helps identify tables that are actively receiving data versus those that have stopped or are delayed. Tables are marked **Fresh** or **Critical** based on a configurable look-back window.

```
EventhouseIngestionResultsLogs
| where Timestamp > ago(20d)
| summarize LastIngestion = max(Timestamp) by TableName
| extend Status = iff(LastIngestion >= ago(7d), "Fresh", "Critical")
| project TableName, LastIngestion, Status
| order by LastIngestion desc
```

2. Table Load Volume Trends

This pattern tracks how many rows are ingested into each table on a daily basis. It helps detect sudden drops, spikes, or irregular data growth. Useful for identifying upstream pipeline issues or unexpected data surges.

```
EventhouseDataOperationLogs
| where Timestamp > ago(25d)
| summarize TotalRows = sum(TotalRowsCount) by bin(Timestamp, 1d), TableName
```

3. Daily Ingestion Status

This view shows daily ingestion activity along with success or failure status. It helps quickly identify days with ingestion errors or reduced activity. Error details provide additional context for troubleshooting.

```
EventhouseIngestionResultsLogs
| where Timestamp > ago(30d)
| summarize Count = count() by bin(Timestamp, 1d), TableName, Status, IngestionErrorDetails
| sort by Timestamp desc
```

4. Schema Change Detection

This pattern monitors schema-related operations executed on Eventhouse tables. It helps detect structural changes such as added, removed, or modified columns. Early visibility into schema changes reduces downstream data breakages.

```
EventhouseQueryLogs
| where Timestamp > ago(30d)
  and QueryText has_any (
        ".alter table", ".create table", ".drop table",
        ".rename table", ".alter column", ".drop column",
        ".add column", ".create-or-alter table"
    )
| project Timestamp, DatabaseName, Identity, QueryText, Status
| sort by Timestamp desc
```

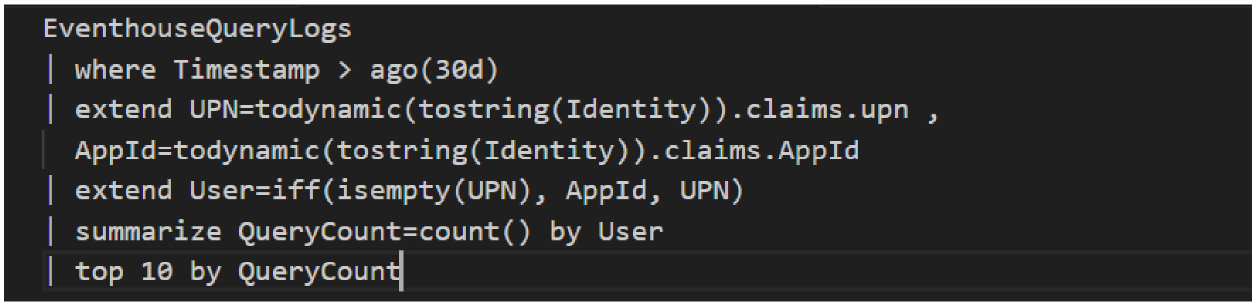
5. Query Volume Over Time

This pattern tracks how query execution volume changes over time. It helps understand usage patterns, peak load periods, and system demand. Useful for performance monitoring and capacity planning.

```
EventhouseQueryLogs
| where Timestamp > ago(30d)
| summarize QueryCount = count() by bin(Timestamp, 1h), Status
| sort by Timestamp desc
```

6. Top Users by Query Count

This pattern identifies the most active users or applications interacting with Eventhouse. It helps with auditing, usage analysis, and identifying heavy consumers. Supports governance and cost optimization initiatives.



```
EventhouseQueryLogs
| where Timestamp > ago(30d)
| extend UPN=todynamic(tostring(Identity)).claims.upn ,
  AppId=todynamic(tostring(Identity)).claims.AppId
| extend User=iff(isempty(UPN), AppId, UPN)
| summarize QueryCount=count() by User
| top 10 by QueryCount
```

7. Memory Usage Over Time

This pattern tracks peak memory usage of commands executed in Eventhouse. It helps identify memory-intensive workloads and potential performance risks. Useful for tuning queries and managing resource utilization.

```
EventhouseCommandLogs
| where Timestamp > ago(30d)
| summarize PeakMemory = max(MemoryPeakBytes) by bin(Timestamp, 1h)
```

8. CPU Usage Over Time

This pattern monitors average CPU consumption across commands over time. It helps detect processing bottlenecks and unusually heavy workloads. Useful for understanding system performance trends.

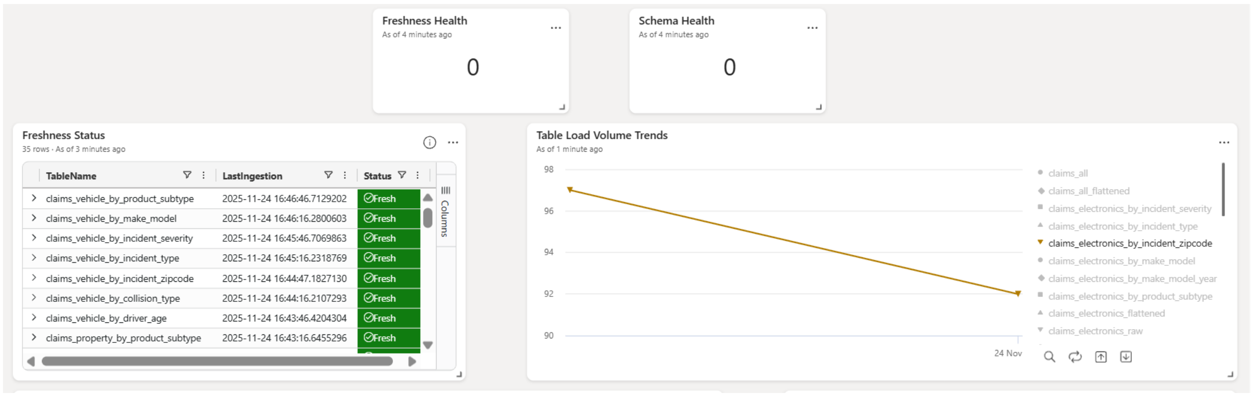
```
EventhouseCommandLogs
| where Timestamp > ago(30d)
| summarize AvgCpuMs = avg(CpuTimeMs) by bin(Timestamp, 1h)
```

9. High CPU Commands

This pattern highlights commands that consume the most CPU resources. It helps quickly identify inefficient or long-running queries. Useful for performance optimization and root cause analysis.

```
EventhouseCommandLogs
| where Timestamp > ago(30d)
| top 10 by CpuTimeMs desc
| project Timestamp, DatabaseName, CommandText, CpuTimeMs, DurationMs, FailureReason, Status
```

Realtime Observability Dashboard



Freshness Health
As of 4 minutes ago
0

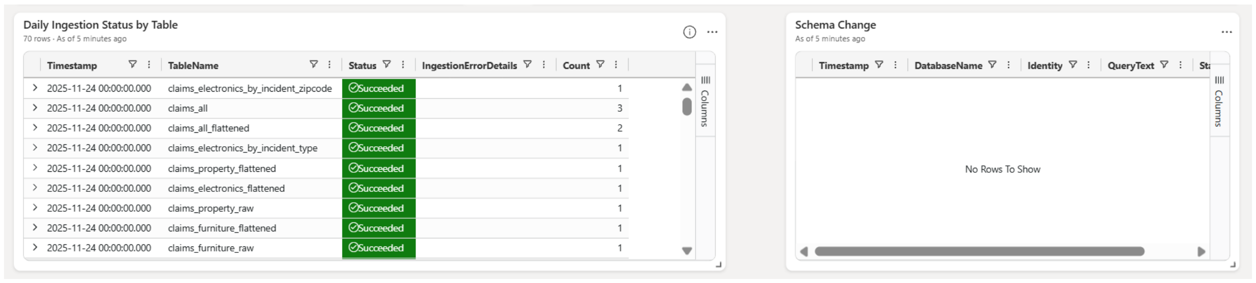
Schema Health
As of 4 minutes ago
0

Freshness Status
35 rows · As of 3 minutes ago

TableName	LastIngestion	Status
claims_vehicle_by_product_subtype	2025-11-24 16:46:46.7129202	Fresh
claims_vehicle_by_make_model	2025-11-24 16:46:16.2800603	Fresh
claims_vehicle_by_incident_severity	2025-11-24 16:45:46.7069863	Fresh
claims_vehicle_by_incident_type	2025-11-24 16:45:16.2318769	Fresh
claims_vehicle_by_incident_zipcode	2025-11-24 16:44:47.1827130	Fresh
claims_vehicle_by_collision_type	2025-11-24 16:44:16.2107293	Fresh
claims_vehicle_by_driver_age	2025-11-24 16:43:46.4204304	Fresh
claims_property_by_product_subtype	2025-11-24 16:43:16.6455296	Fresh

Table Load Volume Trends
As of 1 minute ago

Columns
98
96
94
92
90
24 Nov
claims_all
claims_all_flattened
claims_electronics_by_incident_severity
claims_electronics_by_incident_type
claims_electronics_by_incident_zipcode
claims_electronics_by_make_model
claims_electronics_by_make_model_year
claims_electronics_by_product_subtype
claims_electronics_flattened
claims_electronics_raw



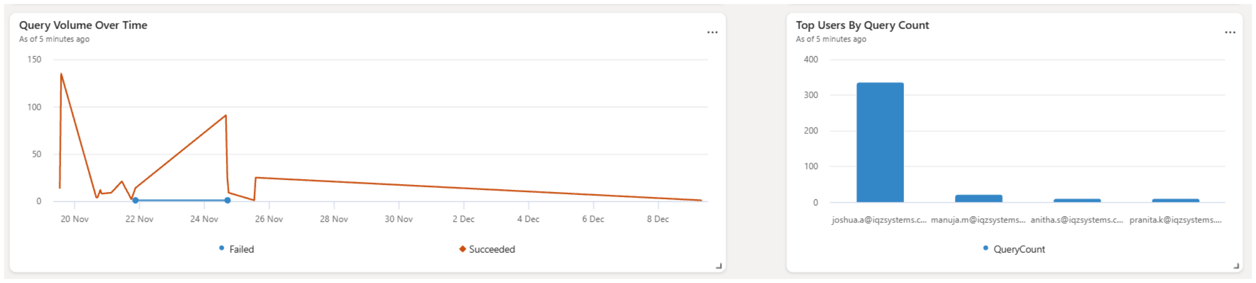
Daily Ingestion Status by Table
70 rows · As of 5 minutes ago

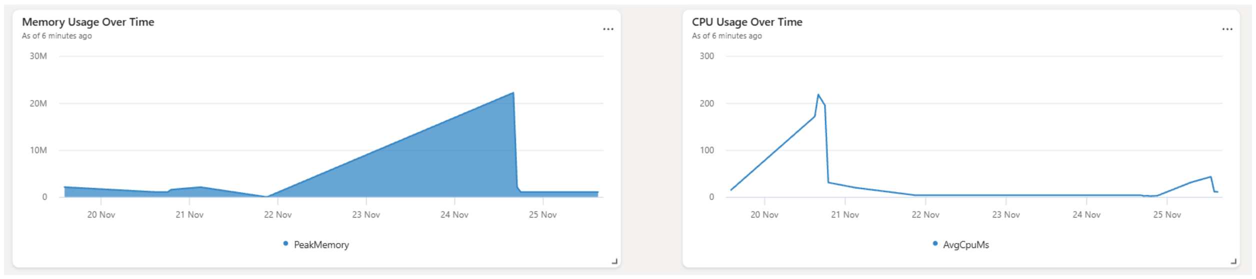
Timestamp	TableName	Status	IngestionErrorDetails	Count
2025-11-24 00:00:00.000	claims_electronics_by_incident_zipcode	✓Succeeded		1
2025-11-24 00:00:00.000	claims_all	✓Succeeded		3
2025-11-24 00:00:00.000	claims_all_flattened	✓Succeeded		2
2025-11-24 00:00:00.000	claims_electronics_by_incident_type	✓Succeeded		1
2025-11-24 00:00:00.000	claims_property_flattened	✓Succeeded		1
2025-11-24 00:00:00.000	claims_electronics_flattened	✓Succeeded		1
2025-11-24 00:00:00.000	claims_property_raw	✓Succeeded		1
2025-11-24 00:00:00.000	claims_furniture_flattened	✓Succeeded		1
2025-11-24 00:00:00.000	claims_furniture_raw	✓Succeeded		1

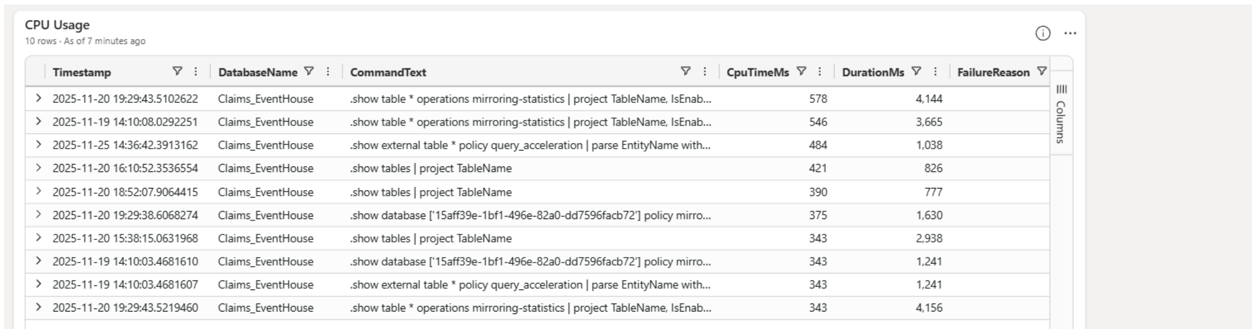
Schema Change
As of 5 minutes ago

Timestamp	DatabaseName	Identity	QueryText	St

No Rows To Show







CPU Usage
10 rows · As of 7 minutes ago

Timestamp	DatabaseName	CommandText	CpuTimeMs	DurationMs	FailureReason
2025-11-20 19:29:43.5102622	Claims_EventHouse	.show table * operations mirroring-statistics | project TableName, IsEnab...	578	4,144	
2025-11-19 14:10:08.0292251	Claims_EventHouse	.show table * operations mirroring-statistics | project TableName, IsEnab...	546	3,665	
2025-11-25 14:36:42.3913162	Claims_EventHouse	.show external table * policy query_acceleration | parse EntityName with...	484	1,038	
2025-11-20 16:10:52.3536554	Claims_EventHouse	.show tables | project TableName	421	826	
2025-11-20 18:52:07.9064415	Claims_EventHouse	.show tables | project TableName	390	777	
2025-11-20 19:29:38.6068274	Claims_EventHouse	.show database ['15aff39e-1bf1-496e-82a0-dd7596facb72'] policy mirro...	375	1,630	
2025-11-20 15:38:15.0631968	Claims_EventHouse	.show tables | project TableName	343	2,938	
2025-11-19 14:10:03.4681610	Claims_EventHouse	.show database ['15aff39e-1bf1-496e-82a0-dd7596facb72'] policy mirro...	343	1,241	
2025-11-19 14:10:03.4681607	Claims_EventHouse	.show external table * policy query_acceleration | parse EntityName with...	343	1,241	
2025-11-20 19:29:43.5219460	Claims_EventHouse	.show table * operations mirroring-statistics | project TableName, IsEnab...	343	4,156	

Difference Between Power BI Dashboard and Real-Time Dashboard

Power BI Dashboard

- Built using tables created from notebook executions.
- Shows historical and batch-processed metrics such as freshness status, volume checks, schema changes, and data quality results.
- Data is refreshed on a scheduled basis, so insights are not instant.
- Used mainly for analysis, reporting, and trend monitoring over time.

Real-Time Dashboard

- Built using KQL queries on Eventhouse real-time monitoring tables.
- Displays near real-time system and ingestion activity such as ingestion status, query volume, schema operations, CPU, and memory usage.
- Updates continuously as events occur.
- Used for live monitoring, operational visibility, and quick issue detection.

Alerts Using Data Activator

To enable proactive monitoring, alerts were configured using Microsoft Fabric Data Activator on top of both the Power BI dashboards and real-time dashboards.

Data Activator continuously monitors the underlying metrics and automatically triggers alerts when defined conditions are met, without requiring manual dashboard checks.

These alerts help in:

- Early detection of data freshness issues
- Identifying abnormal volume changes
- Notifying schema or ingestion failures
- Highlighting critical system or performance issues in real time

Alerts are sent to relevant stakeholders as soon as thresholds are breached, enabling faster investigation and resolution.

Summary

The solution implements a layered approach to data monitoring by capturing observability and data quality metrics directly from notebooks and persisting them as structured tables. These tables act as the foundation for both historical analysis through Power BI dashboards and real-time monitoring through Eventhouse dashboards. By complementing visual insights with automated alerts, the approach ensures data issues are identified early, monitored continuously, and addressed before they impact downstream consumers.