## ****Test Scenario ID: SEC-01 – Data Access Control Enforcement****

### 🔹 ****Purpose****

This scenario ensures that access to sensitive data fields, especially personally identifiable information (PII) or regulated attributes, is **restricted based on user roles**. It validates that only authorized users can view unmasked values, while others receive masked or redacted versions, preserving compliance with data protection policies (e.g., GDPR, internal controls).

### 🔹 ****Scope and Components Involved****

* **Data Zones**:
  + Bronze (raw ingestion)
  + Silver (cleaned and enriched)
  + Consumer Layer (e.g., BI, dashboards, analytics)
* **Sensitive Data Examples**:
  + Customer names, emails, policy numbers, credit scores
* **Roles Covered**:
  + Data Engineers (full access)
  + Business Analysts (restricted access)
  + External auditors or viewers (redacted)

### 🔹 ****Validation Objectives****

* Verify **field-level access policies** are enforced at each layer
* Confirm masking (e.g., full, partial, hashing) is applied based on roles
* Ensure unauthorized users **cannot bypass controls** through SQL, API, or dashboards
* Validate that **audit logs** capture who accessed sensitive fields and when

## 🧪 ****Testing Approach****

### ✅ ****Role-Based Access Simulation****

1. Identify sensitive columns in the Bronze/Silver layer (e.g., customer\_email, user\_ssn)
2. Using three test personas (Engineer, Analyst, Viewer):
   * Attempt to query sensitive fields directly
   * Try accessing views with masked fields (if implemented via Unity Catalog or SQL Views)
   * Attempt exporting data via APIs or dashboards
3. Validate:
   * Full data is accessible only to authorized roles
   * Masking logic (e.g., REPLACE, HASH, NULLIFY) works as expected
   * Unauthorized attempts are either denied or logged

### 📋 ****Validation Methods****

* Role-based SQL query execution using RBAC enforcement
* Field masking checks in query results and dashboards
* Audit logs in Databricks / platform to trace access attempts
* Data quality scan for sensitive values visible to unauthorized roles

## ✅ ****Success Criteria****

* Authorized roles can view sensitive data without restrictions
* Unauthorized users cannot access raw or unmasked PII fields
* Logs show all sensitive data access with timestamps and user identity
* Masking is consistent across direct queries and downstream tools

### 🧾 ****Exclusions****

* Audit trail of access logs is separately covered in **SEC-02**
* Downstream encryption or at-rest security is handled outside test scope (infra-level)

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ10 – Confidentiality** | Ensures sensitive data is protected by access controls |
| **RL DQ06 – Reliability** | Maintains consistent data governance and compliance |
| **RL DQ09 – Traceability** | Enables tracking of who accessed what, when, and how |

### 🔐 ****Pre-requisites & Access Required****

* Defined user roles and test accounts (Engineer, Analyst, Viewer)
* Unity Catalog or RBAC setup (if in Databricks or Snowflake)
* List of classified sensitive fields
* Access to platform audit logs or telemetry data

Let me know if you’d like a templated SQL script for field-level masking or integration steps to configure Unity Catalog policies for sensitive data columns.

##### You said:

SEC-02

###### ChatGPT said:

Here is the detailed technical test scenario for **SEC-02 – Audit Trail of Access Logs**, focused on validating that **all access to data**—especially sensitive or regulated data—is **tracked, timestamped, and auditable** across the ingestion and consumption layers.

## 🔐 ****Test Scenario ID: SEC-02 – Audit Trail of Access Logs****

### 🔹 ****Purpose****

This scenario ensures that every read or write operation performed on the data—especially on PII or critical business fields—is captured with metadata including **user ID, timestamp, action type, and resource accessed**. This is essential for maintaining regulatory compliance, supporting forensic investigations, and ensuring transparency in data usage.

### 🔹 ****Scope and Components Involved****

* **Audit Coverage Targets**:
  + Bronze and Silver Delta tables
  + Views or queries run against PII or customer-sensitive datasets
  + Notebooks, dashboards, SQL endpoints
* **Actors Tracked**:
  + Engineers, Analysts, External Users
  + Automated jobs, pipelines, or service principals
* **Audit Systems**:
  + Databricks audit logs (if enabled)
  + Unity Catalog / Access Control Logs
  + Azure Monitor, Sentinel, or custom log ingestion layer

### 🔹 ****Validation Objectives****

* Confirm that **every access event** is recorded with required metadata
* Ensure logs capture access **to specific columns**, not just general table-level hits
* Validate logs are **immutable, retained, and queryable**
* Ensure logs include **context**: action (READ/WRITE), user, IP/location (if available), tool used (SQL Editor, Notebook, API)

## 🧪 ****Testing Approach****

### ✅ ****Simulate Access via Multiple Interfaces****

1. Query sensitive data using:
   * SQL queries via Databricks SQL Editor
   * Notebooks (Python/Scala)
   * External tools (BI dashboards, JDBC/ODBC connectors)
2. Perform:
   * Read operations (SELECT)
   * Write operations (INSERT, DELETE, UPDATE if permitted)
3. Validate:
   * Each action appears in the audit log with details:
     + User ID
     + Timestamp
     + Operation type
     + Table/column accessed
     + Tool/interface used

### 📋 ****Validation Methods****

* Query native audit logs (e.g., Databricks Audit Logs via Log4j or Azure Diagnostic Settings)
* Compare audit log entries against known actions
* Search for anomalies or gaps (e.g., unlogged access attempts)
* Test log retention period and export policies

## ✅ ****Success Criteria****

* All read/write operations on sensitive data are logged
* Audit logs are complete, immutable, and accessible to authorized auditors
* Metadata in logs is detailed and aligned with compliance needs (e.g., GDPR Article 30, ISO 27001)
* Gaps or tampering in logging are detectable

### 🧾 ****Exclusions****

* Masking or access enforcement logic is tested under **SEC-01**
* Network- or firewall-level access logging is out of scope

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ09 – Traceability** | Enables full lineage and access tracking for each record |
| **RL DQ10 – Confidentiality** | Supports oversight of who accessed sensitive fields |
| **RL DQ06 – Reliability** | Ensures secure and governed access ecosystem is auditable |

### 🔐 ****Pre-requisites & Access Required****

* Audit logging enabled for Databricks / Delta tables (via Unity Catalog, Azure Monitor, etc.)
* Permissions to:
  + Query audit logs
  + Perform controlled data access tests
* List of sensitive assets to focus access simulations on

Let me know if you'd like a JSON schema or sample query format to validate audit logs or a checklist for implementing access logging via Unity Catalog and Azure integration.

##### You said:

BIZ-01

###### ChatGPT said:

Here is the detailed technical test scenario for **BIZ-01 – Rating Score Sanity Range**, which ensures that business-specific logic for numeric values—particularly those generated by ML models or business rules—remain within **expected, valid boundaries**.

## 📊 ****Test Scenario ID: BIZ-01 – Rating Score Sanity Range****

### 🔹 ****Purpose****

This scenario verifies that all calculated or ingested rating scores fall within a **valid, business-approved range**—typically 0 to 100. These scores may originate from pricing engines, ML models, or scoring logic in source systems. The aim is to prevent propagation of **invalid, corrupt, or miscalculated values** that can affect downstream analysis, pricing, or eligibility decisions.

### 🔹 ****Scope and Components Involved****

* **Data Sources**: Messages from Athena, PRECOM, or pricing services
* **Data Field of Interest**: rating\_score, confidence\_score, eligibility\_score
* **Layers Covered**:
  + Bronze (raw values)
  + Silver (validated/enriched values)
  + Analytics / Business Outputs
* **Boundaries to Validate**:
  + Hard range: 0 ≤ score ≤ 100
  + Optional: Minimum signal threshold (e.g., scores > 20 considered valid for use)

### 🔹 ****Validation Objectives****

* Ensure rating values fall within the allowed numeric range
* Detect and quarantine any **out-of-bound scores**
* Identify **negative values, NaNs, or extreme outliers**
* Confirm data pipelines enforce the rule without suppressing records silently

## 🧪 ****Testing Approach****

### ✅ ****Range Check with Simulated and Live Data****

1. Inject test messages with:
   * Valid rating scores (e.g., 45, 99.8)
   * Invalid scores (e.g., -5, 150, NULL, NaN)
2. Ingestion into Bronze should not fail but may flag invalid entries.
3. Silver layer should:
   * Filter or isolate invalid scores
   * Optionally log or move to quarantine table
   * Tag data for reprocessing if threshold breached
4. Create alert conditions for large volume of invalid ratings

### 📋 ****Validation Methods****

* Run queries to detect:
  + Values < 0, > 100, IS NULL, IS NAN
* Check business rule logic (e.g., Spark SQL WHERE clause or validation UDF)
* Review error logs or rejected records dataset (if implemented)
* Confirm that valid ratings are still passing through unaffected

## ✅ ****Success Criteria****

* All invalid rating values are detected and handled appropriately
* No invalid scores pass into analytics or Silver outputs without flagging
* Valid data flow remains uninterrupted
* Logs or rejection tables are updated with offending rows

### 🧾 ****Exclusions****

* ML model performance or scoring algorithm quality is out of scope
* Business thresholds (e.g., eligible vs. high-risk bands) tested in downstream semantic layers

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Validity** | Ensures numeric values comply with logical constraints |
| **RL DQ06 – Reliability** | Blocks corrupt business logic from affecting downstream consumers |
| **RL DQ08 – Relevance** | Guarantees score values are actionable and meaningful |

### 🔐 ****Pre-requisites & Access Required****

* Defined scoring logic or schema documentation from business teams
* Access to:
  + Bronze and Silver tables with rating\_score or equivalent
  + Logs or rule-engine outputs
* Ability to insert controlled test data and query resulting tables

Let me know if you'd like to include this as a rule in your automated data quality framework (e.g., using Great Expectations, Unity Catalog expectations, or PyDeequ).

##### You said:

BIZ-02

###### ChatGPT said:

Here is the detailed technical test scenario for **BIZ-02 – Quote Validity Period Check**, which ensures all quote-related records respect defined business time windows for eligibility and relevance.

## 📅 ****Test Scenario ID: BIZ-02 – Quote Validity Period Check****

### 🔹 ****Purpose****

This scenario validates that each quote ingested or transformed into the Silver layer contains a **valid quote\_timestamp**, and that the quote falls within the **acceptable business time window** (e.g., within the past 30 calendar days). It prevents outdated or future-dated quotes from being processed, displayed, or used for pricing, risk, or reporting decisions.

### 🔹 ****Scope and Components Involved****

* **Data Sources**: MQS message types (e.g., QUOTE\_REQUEST, PRECOM\_RESPONSE)
* **Fields Involved**: quote\_timestamp, valid\_until, event\_time
* **Layers Covered**:
  + Bronze: raw capture of quote events
  + Silver: transformed/filtered quote data
* **Expected Validity Rules**:
  + Quote timestamp should not be more than X days old (e.g., 30)
  + Should not be in the future (compared to system time or ingestion time)
  + Optional: Business-configurable grace window

### 🔹 ****Validation Objectives****

* Verify all quotes fall within expected age range from current time
* Filter or flag quotes with:
  + Timestamps older than 30 days
  + Invalid or missing timestamps
  + Future-dated records
* Validate that rejected quotes are logged or excluded from decisioning

## 🧪 ****Testing Approach****

### ✅ ****Inject Valid and Invalid Quotes****

1. Create test quotes with:
   * Valid timestamp (e.g., within 10 days)
   * Outdated timestamp (e.g., 45 days old)
   * Future-dated timestamp (e.g., 3 days ahead)
   * Null or improperly formatted timestamp
2. Ingest into EventHub and validate:
   * Bronze layer captures raw record with original timestamp
   * Silver logic filters or flags records that don’t meet time-based validity
3. Optionally, simulate ingestion time drift (delayed ingestion) to assess robustness

### 📋 ****Validation Methods****

* Compare quote\_timestamp against system clock and \_ingest\_time
* Use date-difference logic (DATEDIFF, CURRENT\_DATE - quote\_date) in Spark/SQL
* Check if expired quotes are tagged, filtered, or routed to quarantine
* Confirm that only valid quotes appear in downstream datasets

## ✅ ****Success Criteria****

* Quotes older than the configured threshold are excluded from active Silver views
* No future-dated quotes pass into pricing or eligibility logic
* Logs or audit tags capture the exclusion reason
* Time drift or ingestion lag does not falsely exclude valid quotes

### 🧾 ****Exclusions****

* Quote versioning or renewal cycle logic (handled by quoting engine)
* Duplicate quote handling is addressed separately

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Validity** | Ensures quotes comply with expected business time constraints |
| **RL DQ06 – Reliability** | Blocks stale or invalid records from decision workflows |
| **RL DQ08 – Relevance** | Guarantees only current, actionable quotes are considered |

### 🔐 ****Pre-requisites & Access Required****

* Business rules for quote time windows (e.g., 30-day standard)
* Access to:
  + Bronze and Silver quote tables
  + EventHub to push quote payloads for testing
* Ability to validate ingestion and transformation logic (Spark jobs or DLT)

Let me know if you want this logic implemented as a configurable parameter, or if alerts should be triggered when a large number of quotes fall outside the valid window.

##### You said:

BIZ-03

###### ChatGPT said:

Here is the detailed technical test scenario for **BIZ-03 – Correlation Across MQS Messages**, which ensures consistency of identifiers across related message types processed through the pipeline.

## 🔗 ****Test Scenario ID: BIZ-03 – Correlation Across MQS Messages****

### 🔹 ****Purpose****

This scenario verifies that identifiers like QuoteID, CacheID, or CorrelationID remain consistent and correctly linked across related MQS message types—such as QUOTE\_REQUEST, PRECOM\_REQUEST, and PRECOM\_RESPONSE. It ensures proper record matching and relational integrity across asynchronous events within the ingestion pipeline.

### 🔹 ****Scope and Components Involved****

* **Message Types**:
  + QUOTE\_REQUEST
  + PRECOM\_REQUEST
  + PRECOM\_RESPONSE
  + (Optional: other related MQS event types)
* **Key Fields**: QuoteID, CacheID, CorrelationID
* **Data Layers**:
  + Bronze: Raw message ingestion with IDs captured as-is
  + Silver: Logical joins or enrichments applied to connect messages

### 🔹 ****Validation Objectives****

* Ensure the same QuoteID appears consistently across all related message types
* Detect any missing, mismatched, or duplicated identifiers between events
* Validate that joins between message types (e.g., Precom to Quote) are successful and complete
* Ensure no orphan messages exist without their counterpart (e.g., PRECOM\_RESPONSE without a PRECOM\_REQUEST)

## 🧪 ****Testing Approach****

### ✅ ****Simulate Correlated and Uncorrelated Flows****

1. Inject test data via EventHub:
   * A sequence of related messages (same QuoteID, CacheID)
   * Orphan messages (e.g., a PRECOM\_RESPONSE without a PRECOM\_REQUEST)
   * Messages with intentionally mismatched QuoteID or CorrelationID
2. Track:
   * Raw captures in Bronze
   * Joins and enrichments in Silver (e.g., JOIN ON QuoteID)
   * Flagged inconsistencies in lineage, logs, or output tables
3. Validate if system handles missing links gracefully (e.g., no hard failure)

### 📋 ****Validation Methods****

* Run join queries between message types and count successful vs. failed correlations
* Check for:
  + Nulls or mismatches in expected join keys
  + Duplicate identifiers across logically unique events
* Validate downstream metrics (e.g., % matched vs. unmatched messages)
* Optionally test lineage views to trace message flows across the pipeline

## ✅ ****Success Criteria****

* All related messages are correctly correlated by IDs
* Inconsistent or orphan messages are flagged or quarantined
* No loss of valid correlated records due to processing delays
* Logs or monitoring metrics indicate the health of cross-message linkage

### 🧾 ****Exclusions****

* Complex business-specific enrichments (e.g., quote-to-policy linking) beyond ID matching
* External service dependency resolution is out of scope

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ07 – Referential Integrity** | Ensures keys consistently link related records |
| **RL DQ01 – Validity** | Ensures IDs maintain structural and business logic consistency |
| **RL DQ06 – Reliability** | Maintains accurate relationship between event flows |

### 🔐 ****Pre-requisites & Access Required****

* Message structure definitions for all MQS types involved
* EventHub access to inject test messages
* Access to:
  + Bronze and Silver tables for those MQS messages
  + Enrichment or join logic scripts
  + Logs or monitoring dashboards showing correlation status

Let me know if you'd like support building automated tests to track join success ratios or integrate alerts for correlation breakdowns in production.

##### You said:

OBS-01

###### ChatGPT said:

Here is the detailed technical test scenario for **OBS-01 – Processing Latency Metric Validation**, focusing on validating observability and monitoring of end-to-end data flow latency.

## ⏱ ****Test Scenario ID: OBS-01 – Processing Latency Metric Validation****

### 🔹 ****Purpose****

This scenario ensures that **latency metrics**—the time taken for a message to travel from source system (Athena/Mule) to its availability in the Silver layer—are **captured, measured, and monitored** consistently. It provides visibility into pipeline performance and helps detect potential delays, bottlenecks, or service degradation in near real-time.

### 🔹 ****Scope and Components Involved****

* **Message Flow**:
  + Event published to EventHub
  + Bronze ingestion (DLT or streaming job)
  + Silver transformation (DLT or ETL batch job)
* **Timestamps Used for Latency Calculation**:
  + event\_time or created\_timestamp from message payload
  + \_ingest\_time in Bronze layer
  + processed\_time or \_silver\_insert\_time in Silver layer
* **Monitoring Layer**:
  + Custom metrics in Databricks
  + Azure Monitor / Log Analytics / Grafana dashboards
  + Alerts on thresholds (e.g., latency > 5 mins)

### 🔹 ****Validation Objectives****

* Confirm latency is computed as the **difference between event creation and Silver availability**
* Ensure latency metrics are **recorded per message or per batch**
* Validate metrics are **visible on dashboards** and trackable over time
* Check if alerts trigger when latency exceeds expected SLAs

## 🧪 ****Testing Approach****

### ✅ ****Controlled Latency Test with Synthetic Messages****

1. Inject a known message into EventHub with a fixed event\_time
2. Record:
   * \_ingest\_time when it lands in Bronze
   * Final Silver layer insert/update time
3. Calculate:
   * Bronze latency = \_ingest\_time - event\_time
   * Silver latency = silver\_timestamp - \_ingest\_time
   * End-to-end latency = silver\_timestamp - event\_time
4. Compare computed latencies to expected thresholds (e.g., < 2 min for stream, < 15 min for batch)

### 📋 ****Validation Methods****

* Run SQL or Python scripts to compute latency values
* Compare against expected SLA values
* Check monitoring dashboards or logs for metrics like:
  + Average latency per message type
  + Percentile latency (e.g., 95th percentile)
* Simulate pipeline slowdowns and validate alert generation

## ✅ ****Success Criteria****

* Latency is captured for every message type and processing step
* Metrics are available in monitoring dashboards or logs
* Latency consistently remains within SLA (e.g., 5 minutes for Bronze, 10 for Silver)
* Alerts trigger when latency breaches defined thresholds

### 🧾 ****Exclusions****

* Root cause analysis for latency breaches (covered under observability incident triage)
* Metrics visualization formatting

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ06 – Reliability** | Measures timeliness of data flow |
| **RL DQ09 – Traceability** | Enables tracking of record progress through system |
| **RL DQ08 – Relevance** | Ensures data is available when it's still useful |

### 🔐 ****Pre-requisites & Access Required****

* Timestamp fields (event\_time, \_ingest\_time, silver\_timestamp) must be reliably populated
* Access to:
  + Databricks SQL/Spark for latency computation
  + Monitoring system or alerting dashboards
* Configurable thresholds for alert conditions

Let me know if you’d like a metric tracking notebook template or want to integrate latency validation into your automated data observability framework.

##### You said:

OBS-02

###### ChatGPT said:

Here is the detailed technical test scenario for **OBS-02 – Data Volume Anomaly Detection**, focusing on identifying abnormal spikes or drops in message volumes across ingestion stages.

## 📉 ****Test Scenario ID: OBS-02 – Data Volume Anomaly Detection****

### 🔹 ****Purpose****

This scenario ensures that **data volumes** ingested into the pipeline are monitored over time to detect **sudden increases or decreases** in record count per message type. These anomalies may signal issues like upstream system errors, duplication, pipeline failures, or silent data loss. Detecting volume trends ensures reliable pipeline health and data completeness.

### 🔹 ****Scope and Components Involved****

* **Stages Tracked**:
  + EventHub (raw incoming messages)
  + Bronze layer (ingestion confirmation)
  + Silver layer (post-transformation availability)
* **Message Types Monitored**:
  + QUOTE\_REQUEST, PRECOM\_REQUEST, PRECOM\_RESPONSE, etc.
* **Monitoring Window**:
  + Hourly, daily, and weekly aggregates per message type

### 🔹 ****Validation Objectives****

* Establish **baseline volume** for each message type using historical data
* Monitor current ingested volume and detect:
  + Sharp drop (potential loss)
  + Unexpected spike (potential duplication or replay)
* Trigger alerts when deviation exceeds threshold (e.g., 20% deviation from average)

## 🧪 ****Testing Approach****

### ✅ ****Simulated Volume Deviations and Monitoring Checks****

1. Inject known volumes of test messages (e.g., 1,000 QUOTE\_REQUEST, then 100 or 5,000)
2. Track actual counts at:
   * EventHub offset
   * Bronze table record count
   * Silver output count
3. Validate:
   * Metric logs capture correct counts
   * Monitoring logic identifies deviation from expected average
   * Alerts or logs show status

### 📋 ****Validation Methods****

* Use aggregation queries (e.g., GROUP BY message\_type, ingestion\_date) to calculate:
  + Daily record volume
  + % deviation from past 7-day average
* Set up alert logic for:
  + Low volume (< threshold)
  + High volume (> upper bound)
* Confirm these thresholds are tunable per message type

## ✅ ****Success Criteria****

* Volume metrics are captured and tracked per message type and stage
* Alerts fire on anomaly detection beyond defined range (e.g., ±20%)
* No false positives when natural data variance occurs
* Historical trend graphs are available for analysis

### 🧾 ****Exclusions****

* Root cause diagnosis (handled in downstream incident response)
* Downstream data product volume checks (covered under E2E validations)

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ06 – Reliability** | Detects loss or over-ingestion of data |
| **RL DQ09 – Traceability** | Ensures end-to-end visibility of data volumes |
| **RL DQ01 – Validity** | Confirms completeness and reasonability of data flow |

### 🔐 ****Pre-requisites & Access Required****

* Established baseline volume per message type and ingestion stage
* Access to:
  + EventHub metrics (offset logs or Azure Monitor)
  + Bronze/Silver tables for volume querying
  + Metric/alerting dashboard (Grafana, Databricks SQL, Azure Monitor)
* Ability to inject controlled volume for test runs

Let me know if you want help setting up anomaly detection rules in SQL or Python, or configuring monitoring dashboards to visualize volume trends effectively.

##### You said:

OBS-03

###### ChatGPT said:

Here is the detailed technical test scenario for **OBS-03 – Time Window Lag Analysis**, which focuses on validating the time delay between when data is generated and when it becomes available in downstream layers.

## ⌛ ****Test Scenario ID: OBS-03 – Time Window Lag Analysis****

### 🔹 ****Purpose****

This test scenario verifies that the **lag between the event’s original generation time and its availability in the Bronze and Silver layers** is within an acceptable threshold. Monitoring this time lag is critical to ensuring timely decision-making, detecting pipeline slowdowns, and ensuring SLAs for near real-time or daily reporting are met.

### 🔹 ****Scope and Components Involved****

* **Key Timestamps**:
  + event\_time (from source system)
  + \_ingest\_time (in Bronze)
  + processed\_time or silver\_insert\_time (in Silver)
* **Message Sources**: Any message type (e.g., QUOTE\_REQUEST, PRECOM\_RESPONSE)
* **Layers**:
  + Bronze: First landing point in data lake
  + Silver: Transformed and business-usable dataset

### 🔹 ****Validation Objectives****

* Measure time lag between:
  1. event\_time and \_ingest\_time (source to Bronze)
  2. \_ingest\_time and silver\_time (Bronze to Silver)
  3. Full path: event\_time to Silver timestamp
* Confirm that this end-to-end lag remains within configured SLAs (e.g., 10 mins for stream, <1 hour for batch)

## 🧪 ****Testing Approach****

### ✅ ****Inject Test Records with Controlled Timestamps****

1. Insert synthetic or delayed messages into EventHub with predefined event\_time
2. Track:
   * Time when records arrive in Bronze (via \_ingest\_time)
   * Time they are written to Silver
3. Compute lag at each hop and total

### 📋 ****Validation Methods****

* Use SQL or Python queries to calculate:
  + lag\_bronze = \_ingest\_time - event\_time
  + lag\_silver = silver\_insert\_time - \_ingest\_time
  + lag\_total = silver\_insert\_time - event\_time
* Compare these against latency SLAs
* Simulate high-latency conditions to test alert triggering

## ✅ ****Success Criteria****

* 95%+ of messages should meet the latency threshold (configurable)
* Alerts are generated when thresholds are exceeded (e.g., >10 mins for Bronze, >1 hr total)
* Time window lag is consistent across message types unless explained

### 🧾 ****Exclusions****

* Business calendar alignment (e.g., end-of-day cutoffs)
* Downstream batch SLA validation (separate operational SLA monitoring)

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ06 – Reliability** | Ensures timely availability of data |
| **RL DQ09 – Traceability** | Enables tracking of when data was created vs. ingested |
| **RL DQ08 – Relevance** | Ensures data remains useful within its business context |

### 🔐 ****Pre-requisites & Access Required****

* Messages must include an accurate event\_time
* Ingestion framework must capture \_ingest\_time and downstream write times
* Access to:
  + Bronze and Silver timestamp fields
  + Monitoring dashboards or alerting framework (if automated)

Let me know if you'd like a real-time lag monitoring script or to integrate this into a dashboard that displays daily time window performance per message type.

##### You said:

EDGE-01

###### ChatGPT said:

Here is the detailed technical test scenario for **EDGE-01 – Empty Message Handling**, focusing on the ingestion system’s ability to gracefully process or reject blank or whitespace-only payloads.

## 🚫 ****Test Scenario ID: EDGE-01 – Empty Message Handling****

### 🔹 ****Purpose****

This test ensures the data pipeline can **detect, handle, and log** completely empty or whitespace-only messages that may originate from upstream systems (e.g., Athena, MuleSoft). Such messages can cause silent ingestion failures, processing errors, or unintentional data pollution in downstream layers if not explicitly managed.

### 🔹 ****Scope and Components Involved****

* **Input Condition**: Messages with:
  + Empty JSON {}, or completely blank payloads
  + Null root node
  + Whitespace-only content
* **Data Flow**:
  + EventHub → Bronze (raw capture) → Silver (transformed layer)
* **Systems Impacted**:
  + Stream processors
  + Parsing logic
  + DLT or Spark transformation layers

### 🔹 ****Validation Objectives****

* Detect and **reject or quarantine** empty messages before transformation
* Ensure no null or blank records are inserted into Bronze/Silver
* Confirm error logs or quarantine datasets reflect the issue
* Maintain pipeline resilience — processing continues despite invalid input

## 🧪 ****Testing Approach****

### ✅ ****Inject Empty and Malformed Test Messages****

1. Push to EventHub:
   * An empty JSON payload {}
   * Blank string ""
   * Null payload or corrupted binary
2. Observe Bronze table:
   * Does it log the record?
   * Is \_corrupt\_record or equivalent field populated?
3. Check Silver:
   * Ensure empty messages are excluded
   * Review any error-handling logic, logs, or rejected datasets

### 📋 ****Validation Methods****

* Query Bronze table for blank payloads or \_corrupt\_record IS NOT NULL
* Check Silver table contains only valid records
* Verify logs contain parsing or validation errors
* Check alerting system (if configured) for anomaly flags

## ✅ ****Success Criteria****

* Empty/blank messages are detected and handled (e.g., excluded from Silver)
* Bronze may capture them for audit but does not process further
* Logs reflect issue clearly (e.g., "Empty payload dropped")
* Processing continues for valid messages without failure

### 🧾 ****Exclusions****

* Malformed structured data (covered in corrupted payload handling)
* Business rule validation (e.g., null mandatory fields is a separate test)

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Validity** | Ensures payloads contain usable structure |
| **RL DQ06 – Reliability** | Avoids pipeline breakage due to malformed messages |
| **RL DQ09 – Traceability** | Allows auditing of dropped empty records |

### 🔐 ****Pre-requisites & Access Required****

* Message queue (EventHub) access to push test payloads
* Query access to Bronze and Silver tables
* Visibility into pipeline logs or rejection capture tables

Let me know if you'd like to automate this check as part of pre-ingestion validation or include an alert when high volume of blank messages are detected.

##### You said:

EDGE-02

###### ChatGPT said:

Here is the detailed technical test scenario for **EDGE-02 – Extreme Size Payloads**, which validates that the ingestion pipeline can correctly handle very large message payloads without failure or data truncation.

## 📦 ****Test Scenario ID: EDGE-02 – Extreme Size Payloads****

### 🔹 ****Purpose****

This scenario tests the ingestion system’s ability to **process oversized messages**, such as JSON payloads larger than typical expected size (e.g., >1MB). It ensures that large payloads from upstream systems (e.g., Athena or MuleSoft) don’t cause parsing failures, performance degradation, or silent truncation during storage or transformation.

### 🔹 ****Scope and Components Involved****

* **Test Data Condition**:
  + Large nested JSON payloads (1–5 MB)
  + Messages with long arrays, embedded structures, or binary fields encoded as strings
* **Systems in Flow**:
  + EventHub → Bronze → Silver
  + Any parsing logic (e.g., Spark jobs, schema enforcement, Delta Live Tables)

### 🔹 ****Validation Objectives****

* Confirm that large payloads are successfully:
  + Ingested from EventHub
  + Written to Bronze layer without truncation
  + Transformed in Silver layer without failure
* Ensure system remains performant or gracefully handles large payload conditions
* Validate alerts or logs if thresholds are crossed (e.g., row size exceeds soft limits)

## 🧪 ****Testing Approach****

### ✅ ****Inject Oversized Payloads****

1. Generate sample JSON messages with:
   * Large nested data (e.g., large transaction history or policy object)
   * Padding strings to reach 1MB–5MB size
2. Push messages to EventHub
3. Track:
   * Successful capture in Bronze
   * End-to-end latency or failures in transformation jobs
   * Storage footprint and performance metrics
4. Monitor system logs for warnings about row size, memory, or timeouts

### 📋 ****Validation Methods****

* Confirm record exists in Bronze with full payload
* Compare input vs. stored payload (length, field count)
* Check Spark job logs for memory pressure or task retries
* Evaluate metrics such as:
  + Processing time per record
  + Input size per batch
  + Any dropped or corrupted record count

## ✅ ****Success Criteria****

* All large messages are ingested and stored in full
* Silver transformation jobs complete successfully
* No truncation, corruption, or transformation errors
* System metrics reflect expected load without failure

### 🧾 ****Exclusions****

* File-based ingestion (this scenario focuses on EventHub streaming)
* Compression testing (may be included separately)

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Validity** | Ensures large records are fully captured and stored |
| **RL DQ06 – Reliability** | Verifies system remains stable under data pressure |
| **RL DQ09 – Traceability** | Large records remain traceable without truncation |

### 🔐 ****Pre-requisites & Access Required****

* Tools or scripts to generate large JSON test data
* EventHub access to publish oversized test messages
* Bronze/Silver table access for verification
* Logs and Spark job monitoring for runtime diagnostics

Let me know if you'd like a sample payload generator or want to integrate message size tracking into your observability framework.

##### You said:

EDGE-03

###### ChatGPT said:

Here is the detailed technical test scenario for **EDGE-03 – Out-of-Order Event Handling**, focused on validating how the pipeline processes messages that arrive in a non-sequential order.

## 🔄 ****Test Scenario ID: EDGE-03 – Out-of-Order Event Handling****

### 🔹 ****Purpose****

This test ensures that the ingestion pipeline can **handle messages that arrive out of their natural time or event sequence**. This is critical for time-series data, business events like quote lifecycle stages (QUOTE\_REQUEST, PRECOM\_REQUEST, PRECOM\_RESPONSE), and correlated identifiers like QuoteID or CacheID. It prevents data quality issues caused by late or disordered message arrival.

### 🔹 ****Scope and Components Involved****

* **Applicable Event Types**: Any event with a timestamp or sequence dependency  
  (e.g., quote creation before rating, response after request)
* **Time Attributes Considered**:
  + event\_time (logical timestamp)
  + \_ingest\_time (physical ingestion time)
* **Data Flow**: EventHub → Bronze → Silver

### 🔹 ****Validation Objectives****

* Confirm messages are **not rejected** solely due to out-of-order arrival
* Ensure **event-time ordering is respected** during transformation
* Validate that aggregations, joins, or deduplications use **correct event\_time**, not arrival time
* Confirm **late-arriving data** is handled according to business logic or windowing policies

## 🧪 ****Testing Approach****

### ✅ ****Inject Events in Shuffled Order****

1. Prepare 3 messages:
   * PRECOM\_RESPONSE with event\_time = T3
   * QUOTE\_REQUEST with event\_time = T1
   * PRECOM\_REQUEST with event\_time = T2
2. Publish them in **reverse order** (T3, T2, T1)
3. Validate:
   * Bronze captures all messages regardless of order
   * Silver layer processes and stores them **sorted by event\_time**
   * No critical joins fail due to temporal misalignment

### 📋 ****Validation Methods****

* Check for consistent time-based ordering in Silver
* Review joins across message types (e.g., QuoteID matching)
* Confirm no nulls or join misses due to out-of-sequence arrival
* Monitor metrics or logs for "late event" handling

## ✅ ****Success Criteria****

* All out-of-order messages are captured and processed without loss
* Time-based joins or transformations succeed based on event\_time
* No rejected records due to sequence violations
* System respects event-time semantics where required

### 🧾 ****Exclusions****

* Complex business processes depending on SLA deadlines (e.g., windowed aggregations)
* External system impact of delayed responses (not in scope of ingestion test)

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Validity** | Logical consistency of time and sequence across records |
| **RL DQ06 – Reliability** | Ensures robustness of pipeline to real-world event delays |
| **RL DQ09 – Traceability** | Accurate lineage across asynchronous events |

### 🔐 ****Pre-requisites & Access Required****

* Message publishing control (can simulate out-of-order scenarios)
* Access to:
  + Bronze and Silver layers
  + event\_time and \_ingest\_time fields
  + Transformation logic for time-based operations

Let me know if you'd like to create a validation script for time skew detection or integrate out-of-order handling checks in your DLT pipeline logic.

##### You said:

DQ-09-01

###### ChatGPT said:

Here is the detailed technical test scenario for **DQ-09-01 – Lineage Tag Validation**, focused on ensuring full traceability of records from source to target with proper metadata tagging.

## 🧬 ****Test Scenario ID: DQ-09-01 – Lineage Tag Validation****

### 🔹 ****Purpose****

This test verifies that every record processed through the ingestion pipeline contains **metadata tags that support full lineage tracing**—from the originating source system (e.g., Athena or MuleSoft) to the Bronze and Silver layers. These lineage fields are crucial for auditability, debugging, and ensuring accountability in data movement across systems.

### 🔹 ****Scope and Components Involved****

* **Lineage Metadata Fields**:
  + message\_id, source\_system, event\_time, \_ingest\_time, pipeline\_id, batch\_id
* **Traceable Flow**:
  + EventHub (source message)
  + Bronze Layer (raw ingest with metadata)
  + Silver Layer (transformed output with lineage carried forward)

### 🔹 ****Validation Objectives****

* Confirm that key **lineage fields are populated** at each layer
* Validate **consistency of metadata propagation** from EventHub through Bronze to Silver
* Ensure unique identifiers like message\_id enable tracing of records end-to-end
* Verify **logging or lineage systems** (if available) capture this metadata and support searchability

## 🧪 ****Testing Approach****

### ✅ ****Tag Injection and Verification****

1. Publish sample messages to EventHub with known message\_id, event\_time, and source\_system
2. Verify:
   * Bronze records contain these metadata fields
   * Silver records retain the same or mapped versions of these lineage tags
3. Check lineage audit logs or metadata tables for traceability
4. Ensure that each record can be traced using a unique tag (message\_id or event\_uuid)

### 📋 ****Validation Methods****

* SQL queries to check for presence and consistency of:
  + message\_id
  + source\_system
  + event\_time / \_ingest\_time alignment
* Cross-layer joins using message\_id to validate end-to-end linkage
* Query lineage metadata store (if implemented) to confirm discoverability

## ✅ ****Success Criteria****

* All required lineage fields are present in Bronze and Silver
* Metadata values are accurate and match source system message content
* Each record is traceable across layers using message\_id or event\_uuid
* Logs or audit tables allow querying records by lineage tag

### 🧾 ****Exclusions****

* Field-level lineage (e.g., column derivation tracking) – this is schema-level traceability
* Application-level logging outside of pipeline scope

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ09 – Traceability** | Full visibility of data lifecycle through metadata tagging |

### 🔐 ****Pre-requisites & Access Required****

* Ingestion process must extract and retain message\_id, source\_system, event\_time
* Access to:
  + Bronze and Silver datasets
  + Metadata tables or logs
  + EventHub test publisher with message ID control

Let me know if you'd like to automate lineage validation or implement a metadata registry for searching records by tag.

##### You said:

DQ-09-02

###### ChatGPT said:

Here is the detailed technical test scenario for **DQ-09-02 – Data Provenance Check**, focused on verifying that **lineage logs** or metadata capture the full history of a record’s journey through the data pipeline.

## 🧾 ****Test Scenario ID: DQ-09-02 – Data Provenance Check****

### 🔹 ****Purpose****

This scenario ensures that **each data record’s origin, transformation path, and destination** are traceable and documented either within the dataset or through a metadata/audit log system. Data provenance is critical for regulatory compliance, troubleshooting, and validating the integrity of business analytics built on these datasets.

### 🔹 ****Scope and Components Involved****

* **Pipeline Flow Audited**:
  + Source System (Athena, Mule)
  + EventHub (MQ transport)
  + Bronze Layer (raw ingestion with metadata)
  + Silver Layer (transformed, filtered, enriched)
* **Lineage Storage Methods**:
  + Inline metadata columns (e.g., event\_time, message\_id, source\_system, \_ingest\_time, pipeline\_id)
  + Logging/audit tables (e.g., DLT audit logs, Databricks system tables, custom log sink)

### 🔹 ****Validation Objectives****

* Ensure **lineage logs or system tables** capture every message processed
* Validate logs record key transition stages (e.g., ingestion time, transformation time, status)
* Confirm lineage can be queried by key fields like message\_id, source\_system, or timestamp
* Detect any records that were dropped, transformed, or excluded — with audit trail

## 🧪 ****Testing Approach****

### ✅ ****End-to-End Logging Simulation****

1. Inject messages into EventHub with traceable fields (message\_id, event\_time)
2. Track:
   * Presence of record in Bronze
   * Presence and transformation in Silver
   * Entry in lineage logs or metadata table capturing:
     + Step (Bronze ingest, Silver transform)
     + Timestamps
     + Status (success/fail/filtered)
3. Run queries on audit tables to validate:
   * Record flow matches expected lifecycle
   * No silent drops or missing lineage tags

### 📋 ****Validation Methods****

* Join metadata logs to production tables using message\_id
* Confirm completeness of log:
  + Message ingestion logged ✅
  + Message transformation logged ✅
  + Any filtering/exclusion reason logged ✅
* Review sample lineage path for select QuoteID or CacheID to confirm trace

## ✅ ****Success Criteria****

* All stages of pipeline generate traceable log entries for each message
* No messages enter Silver without corresponding Bronze/audit log lineage
* Errors, dropped records, and filtering reasons are clearly logged
* Lineage logs can be queried by time, ID, or source system

### 🧾 ****Exclusions****

* Real-time visualization dashboards (this test focuses on data completeness, not UI)
* Downstream data product logs (focus is up to Silver)

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ09 – Traceability** | Ensures full lineage logs exist and are queryable |

### 🔐 ****Pre-requisites & Access Required****

* Lineage logging mechanism enabled (DLT audit log, system tables, etc.)
* Ingestion pipeline includes metadata tagging logic
* Access to:
  + EventHub publishing
  + Bronze and Silver tables
  + Lineage/audit logs or metadata sinks

Let me know if you’d like support setting up a test framework that automatically reconciles pipeline logs against the data tables for complete provenance validation.

##### You said:

DQ-07-01

###### ChatGPT said:

Here is the detailed technical test scenario for **DQ-07-01 – Foreign Key Reference Check**, focusing on validating **referential integrity** across related datasets in the pipeline.

## 🔗 ****Test Scenario ID: DQ-07-01 – Foreign Key Reference Check****

### 🔹 ****Purpose****

This scenario ensures that **foreign key relationships (e.g., QuoteID, CacheID, ProductID)** across records are correctly maintained between related data entities. It helps guarantee the **integrity and completeness** of the dataset, especially in cases where multiple messages or tables must logically reference the same master or transaction data.

### 🔹 ****Scope and Components Involved****

* **Key Fields for Validation**:
  + QuoteID – shared between QUOTE\_REQUEST, PRECOM\_REQUEST, and PRECOM\_RESPONSE
  + CacheID – reused in downstream matching and enrichment
  + Any external or lookup key (e.g., ProductID, CustomerID)
* **Tables/Entities**:
  + Bronze: Captures all raw events
  + Silver: Transformed tables representing business entities (quotes, pricing, responses)

### 🔹 ****Validation Objectives****

* Confirm that all foreign key references in a table **exist in the corresponding parent or master table**
* Detect and report **orphan records** that refer to a non-existent QuoteID or CacheID
* Validate consistency of key values across message types and event versions

## 🧪 ****Testing Approach****

### ✅ ****Cross-Entity Join Validation****

1. Collect unique IDs from Silver tables (e.g., PRECOM\_RESPONSE.QuoteID)
2. Join them with the expected source tables (e.g., QUOTE\_REQUEST.QuoteID)
3. Identify:
   * Missing references (IDs present in child but not in parent)
   * Duplicated keys where uniqueness is expected
4. Optionally, simulate a test with injected messages missing upstream keys

### 📋 ****Validation Methods****

* SQL anti-joins or left joins to find unmatched references
  + e.g., SELECT QuoteID FROM response LEFT JOIN request USING (QuoteID) WHERE request.QuoteID IS NULL
* Distinct count comparison of IDs across tables
* Alerts/logging if foreign key mismatch exceeds threshold

## ✅ ****Success Criteria****

* All foreign keys in downstream entities exist in source/master entities
* No orphaned records in Silver layer (or are explicitly flagged and quarantined)
* Cross-entity joins succeed without nulls on referenced fields

### 🧾 ****Exclusions****

* Soft references where absence is acceptable (e.g., optional associations)
* External data integrity checks outside of the ingestion scope

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ07 – Referential Integrity** | Ensures keys used across datasets have valid parent relationships |

### 🔐 ****Pre-requisites & Access Required****

* Access to all participating datasets: Silver tables for Quote, Precomp, Rating, etc.
* Documentation of expected relationships (mapping doc or business rules)
* Tooling to execute joins and analyze mismatches

Let me know if you'd like to build a reusable SQL validation framework to automate referential integrity checks between evolving message types.

##### You said:

DQ-07-02

###### ChatGPT said:

Here is the detailed technical test scenario for **DQ-07-02 – Cross-Dataset Correlation**, which validates that **key business identifiers remain consistent** across related datasets and layers within the pipeline.

## 🔁 ****Test Scenario ID: DQ-07-02 – Cross-Dataset Correlation****

### 🔹 ****Purpose****

This test ensures that **critical business identifiers** such as QuoteID, CacheID, ProductID, etc., are **consistently carried and matched** across all relevant datasets—from raw messages in Bronze to transformed business entities in Silver. Cross-dataset correlation is key to maintaining semantic consistency in analytics, enrichment, and reporting.

### 🔹 ****Scope and Components Involved****

* **Key Identifiers** for correlation:
  + QuoteID – ties together request, rating, and precomp messages
  + CacheID – links between real-time cache and downstream usage
  + ProductID, RatingGroupID – used for enrichment and pricing logic
* **Target Datasets**:
  + Bronze layer (raw MQS events)
  + Silver layer (Quote Summary, Precomp Detail, Rating Result, etc.)
* **Message Types**:
  + QUOTE\_REQUEST, PRECOM\_REQUEST, PRECOM\_RESPONSE, RATING\_RESULT

### 🔹 ****Validation Objectives****

* Ensure that key IDs **appear consistently** across all message types referencing the same business event
* Validate that **relationships are preserved** through transformations
* Detect and flag records where correlation fails (e.g., a PRECOM\_RESPONSE exists for a QuoteID not present in QUOTE\_REQUEST)

## 🧪 ****Testing Approach****

### ✅ ****Cross-Layer Consistency Check****

1. Extract records from Bronze for a given test window
2. Identify QuoteID, CacheID, and ProductID values per message type
3. Compare corresponding entries in Silver:
   * Verify each QuoteID in PRECOM\_RESPONSE exists in QUOTE\_REQUEST
   * Ensure ProductID matches across enrichment layers
   * Confirm CacheID aligns with the correct rating path
4. Check data lineage or transformation logs (if available) for ID mismatch issues

### 📋 ****Validation Methods****

* Perform inner joins or anti-joins across datasets using QuoteID, CacheID, etc.
* Group by ID and count occurrences to detect:
  + Unexpected duplicates
  + Missing links
* Validate if ID propagation logic (in mapping/transformation rules) is correctly applied

## ✅ ****Success Criteria****

* All key identifiers are **present and correctly matched** across datasets
* No unlinked records are found in Silver
* Consistency maintained even if records arrive out of order (event-time based)

### 🧾 ****Exclusions****

* Soft business joins where correlation is not strictly required
* Validation of field derivation logic (covered in transformation test cases)

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ07 – Referential Integrity** | Ensures key business identifiers are consistently represented across related datasets |

### 🔐 ****Pre-requisites & Access Required****

* Data mapping documentation showing expected cross-entity key relationships
* Access to Bronze and Silver tables
* Tools for executing cross-dataset joins or comparisons (e.g., Databricks notebooks)

Let me know if you'd like help automating this test across all message types or developing visual lineage tools to trace identifier propagation through the layers.

##### You said:

DQ-08-01

###### ChatGPT said:

Here is the detailed technical test scenario for **DQ-08-01 – Stakeholder Review Pipeline**, which ensures the business relevance and usability of the processed data through human validation.

## 🧾 ****Test Scenario ID: DQ-08-01 – Stakeholder Review Pipeline****

### 🔹 ****Purpose****

This test scenario focuses on **validating the business relevance of data outputs** by incorporating regular **reviews with data owners or stakeholders**. While automated data validation ensures structure and consistency, stakeholder reviews help confirm that the **transformed Silver data meets real-world business expectations**, aligns with downstream needs, and supports decision-making.

### 🔹 ****Scope and Components Involved****

* **Stakeholders**:
  + Business analysts
  + Pricing owners
  + Data product managers
* **Target Datasets**:
  + Silver layer tables (e.g., Quote Summary, Precomp Aggregates, Rating Decisions)
* **Focus Areas**:
  + Business logic accuracy (e.g., quote validity, rating thresholds)
  + Timeliness and completeness of data
  + Interpretation of flags, indicators, or calculated fields

### 🔹 ****Validation Objectives****

* Confirm with stakeholders that Silver datasets are **fit-for-purpose** for reporting, pricing, or analytics
* Identify and address **semantic or contextual mismatches** between expected and actual outputs
* Establish a **feedback loop** to capture business-driven corrections or enhancement needs

## 🧪 ****Testing Approach****

### ✅ ****Stakeholder Validation Workflow****

1. Select a **representative sample** of Silver layer data (e.g., 100 recent quotes or precom results)
2. Present key columns to stakeholders in a **review-friendly format** (e.g., Excel, dashboard)
3. Stakeholders verify:
   * Logical correctness of data (e.g., quote expired after 30 days, score in expected range)
   * Accurate mappings and business rule applications
   * Any outliers or anomalies
4. Capture feedback and classify findings as:
   * Confirmed ✔️
   * Needs fix ❌
   * Requires clarification 🔍

### 📋 ****Validation Methods****

* Manual review checklist for business logic
* Structured feedback collection form
* Ticket creation for any gaps or data logic mismatches
* Version-controlled sign-off log for audit

## ✅ ****Success Criteria****

* Business stakeholders **approve the accuracy and usability** of Silver layer data
* No major misinterpretations or logic issues found
* Feedback is documented and acted upon in subsequent transformations

### 🧾 ****Exclusions****

* Automated validations (covered in system testing)
* UI/dashboard issues unless directly related to data interpretation

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ08 – Relevance** | Validates that data meets user expectations and supports business use cases |

### 🔐 ****Pre-requisites & Access Required****

* Clean Silver dataset for sampling
* Clear data dictionary and transformation rules for reference
* Stakeholder availability and context briefing
* Secure review environment (e.g., read-only dashboards or export)

Let me know if you’d like a stakeholder review checklist template or a dashboard prototype to simplify the sign-off process.

##### You said:

MON-01

###### ChatGPT said:

Here is the detailed technical test scenario for **MON-01 – Pipeline Health Monitoring**, which ensures that key metrics about the ingestion and transformation pipeline are continuously tracked and visible.

## 🩺 ****Test Scenario ID: MON-01 – Pipeline Health Monitoring****

### 🔹 ****Purpose****

This test validates that the data pipeline’s operational health is **actively monitored** using measurable metrics. These include data volume, error rates, and processing delays. Real-time monitoring ensures quick detection of issues like ingestion failures, data drop-offs, or transformation bottlenecks, which can otherwise silently impact business reports or downstream applications.

### 🔹 ****Scope and Components Involved****

* **Pipeline Layers**:
  + EventHub (source stream)
  + Bronze Layer (raw ingestion)
  + Silver Layer (transformed output)
* **Monitoring Targets**:
  + Ingested record count per time window
  + Failed record count or quarantine volume
  + Lag between event\_time and \_ingest\_time
  + Pipeline status (running, error, delayed)
* **Monitoring Tools**:
  + Azure Monitor / Log Analytics
  + Databricks metrics dashboard
  + Custom telemetry / observability tools

### 🔹 ****Validation Objectives****

* Ensure key metrics are being emitted and captured by monitoring systems
* Confirm alerts are set up for volume drops, latency spikes, and errors
* Validate dashboards show real-time health of ingestion and transformation

## 🧪 ****Testing Approach****

### ✅ ****Simulated Load & Health Observability****

1. Simulate normal message flow through EventHub → Bronze → Silver
2. Validate that expected metrics (record count, processing time, failures) appear on dashboards
3. Inject test cases to simulate:
   * Sudden drop in message count
   * Spike in ingestion delay
   * Transformation failure
4. Confirm:
   * Metrics reflect anomalies in near real-time
   * Alerts (email, Slack, or portal) are triggered based on thresholds
   * Logs contain actionable error details

### 📋 ****Validation Methods****

* Review and snapshot ingestion dashboards during test events
* Validate alert rules and notification routing
* Compare actual vs. expected metric values under different load conditions
* Confirm time granularity is sufficient (e.g., 5-min buckets)

## ✅ ****Success Criteria****

* All relevant health metrics are emitted and visualized
* Dashboards accurately reflect current state of data flow
* Alerts are triggered when thresholds are crossed (e.g., drop in event volume, spike in errors)
* Logs or telemetry provide sufficient diagnostics for recovery

### 🧾 ****Exclusions****

* Application-level business logic alerts (covered under stakeholder testing)
* UI layout or customization (focus is metric availability and behavior)

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL MON-01 – Observability & Monitoring** | Ensures system emits and displays key operational metrics for transparency |

### 🔐 ****Pre-requisites & Access Required****

* Active ingestion pipeline running through EventHub → Bronze → Silver
* Monitoring dashboards in place (Databricks, Azure Monitor, etc.)
* Access to log streams and alerting configuration
* Controlled ability to inject test events and simulate failures

Let me know if you'd like help building custom health monitoring dashboards or simulating synthetic data flows to test observability under various failure modes.

##### You said:

MON-02

###### ChatGPT said:

Here is the detailed technical test scenario for **MON-02 – Alert Threshold Validation**, which ensures that meaningful alerts are defined, triggered, and actionable when key pipeline thresholds are breached.

## 🚨 ****Test Scenario ID: MON-02 – Alert Threshold Validation****

### 🔹 ****Purpose****

This test ensures that **alerts are configured correctly** across the ingestion pipeline to detect anomalies such as sudden spikes in null values, unexpected drops in record volumes, or elevated error rates. Proper alerting enables **proactive response to issues** before they impact downstream consumers or analytics.

### 🔹 ****Scope and Components Involved****

* **Layers Covered**:
  + EventHub → Bronze → Silver
* **Alert Types**:
  + Volume anomaly (e.g., record count drop > 50%)
  + Null surge in critical fields
  + Error spike in transformation jobs
  + Delays between event time and ingest time (data lag)
* **Alert Delivery**:
  + Email, Teams/Slack, Azure Alerts, Databricks alerts

### 🔹 ****Validation Objectives****

* Confirm **threshold-based alerts** are defined for key pipeline health metrics
* Simulate breaches to **trigger real alerts**
* Validate alert content is clear, actionable, and delivered to the right team

## 🧪 ****Testing Approach****

### ✅ ****Threshold Breach Simulation****

1. Identify alerting rules (e.g., “less than 10,000 messages/hour”)
2. Induce test conditions:
   * Reduce EventHub flow or pause upstream message injection
   * Inject data with intentional nulls in key fields
   * Simulate transformation failures (e.g., schema mismatch)
3. Confirm:
   * Alert is fired within expected detection window
   * Notification reaches designated recipients
   * Logs or dashboards provide context for diagnosis

### 📋 ****Validation Methods****

* View alert history and notifications
* Confirm alert metadata:
  + Trigger time
  + Metric value vs. threshold
  + Pipeline/job affected
* Cross-check logs to validate real cause vs. alert message
* Confirm suppression logic prevents alert fatigue if designed

## ✅ ****Success Criteria****

* All defined thresholds are tested and trigger alerts as expected
* Alerts contain clear descriptions, context, and next-step guidance
* No false negatives (breach not alerted)
* No excessive false positives without valid conditions

### 🧾 ****Exclusions****

* Manual stakeholder alerts (covered under DQ-08-01)
* Alert dashboards or front-end UI monitoring customization

### 🗂 ****Mapped Data Quality Requirements****

| **DQ Ref** | **Description** |
| --- | --- |
| **RL MON-02 – Alert Threshold Validation** | Ensures data and operational anomalies are automatically detected and alerted |

### 🔐 ****Pre-requisites & Access Required****

* Defined alert thresholds and access to alert configuration
* EventHub message injection rights (to simulate volume changes)
* Access to Azure Monitor, Databricks, or other telemetry tools
* Notification system (email/Slack) integration tested