**Test Scenario ID: EH-BR-08 – EventHub Message Type Validation**

**🔹 Purpose**

This test ensures that only **valid and expected message types** (e.g., QUOTE\_REQUEST, PRECOM\_REQUEST, RATING\_RESPONSE) are received from EventHub and logged into the Bronze layer. Any messages that deviate from the expected types must be either filtered, flagged, or routed to a reject log. This is critical for schema enforcement and downstream consistency.

**🔹 Scope and Components Involved**

* **Source**: EventHub (originating from MQS or Athena)
* **Target**: Bronze Layer
* **Expected Types** (based on mapping and screenshots):
  + QUOTE\_REQUEST
  + PRECOM\_REQUEST
  + PRECOM\_RESPONSE
  + RATING\_REQUEST
  + RATING\_RESPONSE
  + STEP\_RESPONSE
  + Possibly others defined in source MQS schemas
* **Reference Artifacts**:
  + Source message mapping specs (e.g., Athena → MQS → EventHub)
  + Event schema or RAML contracts
  + Ingestion transformation logic (DLT or structured streaming code)

**🔹 Validation Objectives**

* Ensure only whitelisted event\_type values are processed and logged into the Bronze table.
* Identify and handle messages with:
  + Unknown or unexpected event\_type
  + Missing event\_type field
  + Typoed or malformed event types (e.g., QUOTE\_REQST)
* Confirm that such records are:
  + Logged separately (quarantine/reject table), or
  + Flagged using is\_valid = false, or
  + Excluded from downstream layers

**🧪 Testing Approach**

**✅ Inject Controlled Event Types into EventHub**

1. Send test messages with:
   * Valid event types (e.g., QUOTE\_REQUEST, PRECOM\_RESPONSE)
   * Invalid or unknown types (e.g., FAKE\_EVENT, 123, null)
2. Validate the result in Bronze:
   * Confirm presence of valid messages
   * Confirm absence (or flagging/logging) of invalid message types
3. Monitor logs or metrics to ensure anomalies are caught and flagged
4. Optionally, validate schema drift is not triggered by invalid types.

**📋 Validation Methods**

* Bronze Layer Validation Queries:
  + SELECT DISTINCT event\_type FROM bronze → Should match predefined list
  + SELECT \* FROM bronze WHERE event\_type NOT IN ('QUOTE\_REQUEST', ..., 'STEP\_RESPONSE') → Should return 0 or expected rejections
* Check reject log/quarantine table (if implemented)
* Validate expectation rules:
  + expect(event\_type IN [...]) in DLT

**✅ Success Criteria**

* Only messages with valid event\_type values are accepted into Bronze.
* Invalid or unknown event types are excluded, logged, or flagged without halting ingestion.
* Silver layer is never impacted by rogue event types.
* All expected event types appear consistently for downstream processing.

**🧾 Exclusions**

* Schema conformity per event type is validated in **EH-BR-02**.
* Message receipt and ingestion trace are covered under **EH-BR-01** and **EH-BR-07**.

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ02 – Conformity** | Enforces strict control over allowed values in event\_type. |
| **RL DQ05 – Validity** | Ensures message types are logically valid and align with schema. |
| **RL DQ09 – Traceability** | Supports debugging of rogue/unexpected events in ingestion path. |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Bronze layer schema and records
  + EventHub message publishing or a test EventHub
  + Mapping documents or MQS schema references
* Capability to:
  + Send test messages via Postman, Kafka tool, or EventHub CLI
  + Track logged messages and ingestion metrics

**🔍 Test Scenario ID: EH-BR-09 – Event Payload Size Validation**

**🔹 Purpose**

This scenario ensures that all EventHub messages ingested into the Bronze layer are within acceptable **payload size limits** as defined by system constraints or business rules. Oversized messages may cause ingestion failures, truncation, or processing bottlenecks, and should be either rejected, logged, or specially handled.

**🔹 Scope and Components Involved**

* **Source**: EventHub (receiving from MQS/Mule/Athena)
* **Target**: Bronze Layer (raw table with \_raw\_payload, message\_id, etc.)
* **Payload Considerations**:
  + Azure EventHub limit is 1 MB per message
  + Business rule threshold may be lower (e.g., 500 KB)
  + Binary fields, embedded documents, or long JSON strings can inflate size
* **Handling strategy**:
  + Reject or quarantine large messages
  + Truncate safely with warning
  + Decompress if using gzip encoding (and validate decoded size)

**🔹 Validation Objectives**

* Validate that messages larger than the configured threshold are:
  + **Not ingested** or
  + Logged as rejected
  + Flagged in Bronze with is\_oversized = true or similar tag
* Ensure downstream layers (Silver) are not impacted by oversized messages
* Prevent pipeline failures due to payload overflow or deserialization errors

**🧪 Testing Approach**

**✅ Controlled Oversize Payload Injection**

1. Publish test messages with:
   * Normal size (e.g., < 100 KB)
   * Borderline size (e.g., 900 KB)
   * Oversized (e.g., > 1 MB uncompressed JSON)
2. Observe behavior at ingestion:
   * Does message land in Bronze or get blocked?
   * Is an error captured in logs?
   * Is oversized record flagged or routed to invalid table?
3. If pipeline allows message compression:
   * Inject large gzip-encoded payload and validate decompression size handling

**📋 Validation Methods**

* Query Bronze layer:
  + SELECT LENGTH(\_raw\_payload) AS size\_bytes FROM bronze ORDER BY size\_bytes DESC
  + Flag any record where size\_bytes > expected\_threshold
* Monitor EventHub or ingestion pipeline logs for:
  + Message drop/retry due to size
  + Deserialization failures
* Check reject table (if implemented) for logged oversize events

**✅ Success Criteria**

* Messages exceeding size threshold are handled gracefully.
* No oversized payload causes ingestion or processing failure.
* Oversize filtering strategy (reject, quarantine, flag) is consistently applied.
* All standard-size messages are ingested without error or truncation.

**🧾 Exclusions**

* Empty or corrupted payload validation is covered under **EH-BR-04**.
* Message count and traceability logic is addressed in **EH-BR-01** and **EH-BR-07**.

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ05 – Validity** | Ensures payloads are within acceptable technical boundaries. |
| **RL DQ06 – Reliability** | Prevents ingestion failures due to unanticipated payload sizes. |
| **RL DQ09 – Traceability** | Allows identification and review of oversize message attempts. |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Bronze layer data with \_raw\_payload or equivalent
  + Azure EventHub metrics or ingestion logs
  + System configuration for payload limits
* Capability to:
  + Send oversized messages to EventHub via test tools or scripts
  + Monitor log streams or pipeline alerts for ingestion errors

**🔍 Test Scenario ID: BR-SL-09 – Derived Field Accuracy Validation**

**🔹 Purpose**

This test ensures that **derived fields created during Bronze-to-Silver transformation** (e.g., valid\_to\_date, is\_test\_event, channel\_group) are **computed accurately according to business logic**. These derived fields often influence downstream decisions and reports, making their correctness essential to data trustworthiness.

**🔹 Scope and Components Involved**

* **Source**: Bronze Layer (raw message + metadata)
* **Target**: Silver Layer (cleaned + derived columns)
* **Examples of Derived Fields**:
  + is\_test\_event based on event\_type or source\_env
  + channel\_group derived from channel\_code
  + valid\_to\_date derived by adding TTL to event\_time
  + event\_source\_category computed from source\_system
* **Reference Artifacts**:
  + Business transformation rules (documentation or tickets)
  + Mapping specs or DLT logic scripts
  + Expected value matrix (mapping table, if applicable)

**🔹 Validation Objectives**

* Ensure all derived fields in Silver are:
  + Present and populated correctly
  + Computed as per documented transformation logic
  + Consistent across similar records (no rule mismatch)
* Validate logic paths such as:
  + Conditional assignment (e.g., IF channel\_code LIKE 'ABC%' THEN 'Partner')
  + Temporal derivation (e.g., expiry = timestamp + 30 days)
  + Default fallback values when source data is null or missing

**🧪 Testing Approach**

**✅ Run Controlled Derivation Test Cases**

1. Create test records in Bronze with:
   * Known channel\_code, event\_type, source\_system, event\_time
2. Trigger DLT transformation into Silver
3. Validate:
   * Derived field values match the expected output
   * Edge cases (null input, unmapped codes) follow fallback rules
4. Negative testing:
   * Provide conflicting/malformed source values and confirm derivation logic holds

**📋 Validation Methods**

* Query derived field logic in Silver:
  + SELECT channel\_code, channel\_group FROM silver and compare to rule set
  + SELECT event\_time, valid\_to\_date FROM silver WHERE valid\_to\_date <> event\_time + INTERVAL '30' DAY
  + SELECT \* FROM silver WHERE is\_test\_event = true AND event\_type NOT LIKE '%TEST%' (logic validation)
* If mappings are driven by lookup tables:
  + Join with mapping table and verify joins are correct
* Use DLT expect clauses or assertions for logic checks

**✅ Success Criteria**

* All derived fields in Silver are correctly computed from Bronze fields.
* Derived values match business logic across various conditions.
* No incorrect, null, or missing derived values are observed unless allowed by design.
* Transformations are repeatable and deterministic.

**🧾 Exclusions**

* Field-level type casting is handled in **BR-SL-04**.
* Business rule filtering and enforcement is covered in **BR-SL-05**.
* Enrichment from external sources is considered out of scope here (unless internal).

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Accuracy** | Ensures correctness of derived business fields. |
| **RL DQ02 – Conformity** | Validates logical transformation rules are followed. |
| **RL DQ05 – Validity** | Checks that output values are within acceptable bounds. |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Bronze and Silver layers
  + Business logic documentation or field transformation specs
  + Mapping tables (if used in derivation)
* Ability to:
  + Load synthetic data with predictable characteristics
  + Inspect transformation logic within DLT or SQL pipelines

**🔍 Test Scenario ID: BR-SL-10 – Silver Layer Field Standardization Validation**

**🔹 Purpose**

This scenario ensures that **standardization rules** (e.g., formatting, casing, date formats, null handling) are consistently applied to data fields in the Silver layer. It is essential to enforce uniform field representations across all records to support reliable querying, analytics, and downstream consumption.

**🔹 Scope and Components Involved**

* **Source**: Bronze Layer (raw fields with various formats)
* **Target**: Silver Layer (cleaned and standardized fields)
* **Field Examples**:
  + event\_type → standardized to UPPERCASE
  + country\_code → trimmed and uppercased
  + event\_date → formatted to yyyy-MM-dd
  + status\_flag → mapped from short values like 'Y'/'N' to 'Active'/'Inactive'
* **Standardization Examples**:
  + Trimming whitespace
  + Enforcing lowercase or uppercase
  + Padding values (e.g., product\_id = 8-digit string)
  + Date/time format consistency
  + Standard null representations (e.g., NULL vs. empty string)
* **Reference Artifacts**:
  + Data mapping or transformation logic
  + Field dictionary or domain rulebook
  + DLT transformation scripts

**🔹 Validation Objectives**

* Confirm that all target fields in Silver layer:
  + Follow defined formatting and data type rules
  + Present uniform structure for similar records
* Validate that no legacy formats, inconsistent casing, or unexpected values persist
* Ensure nulls and empty fields are handled consistently as per policy

**🧪 Testing Approach**

**✅ Execute Format Compliance Tests**

1. Ingest a controlled test set into Bronze with deliberately inconsistent values:
   * Mixed casing, extra spaces, non-standard date formats
2. Run Bronze-to-Silver transformation pipeline
3. Query and validate Silver outputs:
   * All fields match target formatting rules
   * No residual formatting anomalies remain
4. Include boundary values:
   * Null inputs, max/min length strings, invalid enum cases

**📋 Validation Methods**

* Sample validations:
  + SELECT event\_type FROM silver WHERE event\_type <> UPPER(event\_type)
  + SELECT country\_code FROM silver WHERE LENGTH(TRIM(country\_code)) <> LENGTH(country\_code)
  + SELECT event\_date FROM silver WHERE event\_date NOT LIKE '\_\_\_\_-\_\_-\_\_'
  + Validate status\_flag values only in ('Active', 'Inactive')
* Use DLT expect() expressions for real-time checks
* Compare Silver field values to formatting rules in metadata

**✅ Success Criteria**

* 100% compliance with field formatting and standardization logic
* No non-standard casing, spacing, null representation, or unexpected values
* Silver layer is immediately consumable by downstream users or apps without preprocessing

**🧾 Exclusions**

* Type casting and value transformation validations are covered in **BR-SL-04** and **BR-SL-02**
* Derived field validations (e.g., logical derivation) are in **BR-SL-09**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ02 – Conformity** | Fields follow standard definitions, formats, and shapes. |
| **RL DQ05 – Validity** | Values conform to expected data domains or formats. |
| **RL DQ06 – Reliability** | Ensures consistent data appearance and usage across pipeline runs. |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Silver layer and Bronze source data
  + Business rules or domain dictionary for format rules
* Ability to:
  + Inject formatted and misformatted test records
  + Inspect transformation logic or DLT job definition

**🔍 Test Scenario ID: E2E-01 – End-to-End Count Match**

**🔹 Purpose**

This test validates that the **record count remains consistent** as data moves from **EventHub → Bronze → Silver** within a defined processing window. It ensures **no data loss or duplication** occurs throughout the ingestion and transformation stages and provides confidence in pipeline integrity.

**🔹 Scope and Components Involved**

* **Source**: EventHub (message publisher)
* **Intermediary**: Bronze Layer (raw message capture)
* **Target**: Silver Layer (filtered/structured data)
* **Processing Modes**: Micro-batch or streaming (via Delta Live Tables)
* **Expected Message Types**:
  + QUOTE\_REQUEST, PRECOM\_REQUEST, RATING\_RESPONSE, etc.
* **Reference Artifacts**:
  + DLT pipeline lineage
  + Bronze/Silver layer data schema with audit fields like \_ingest\_time, load\_dt
  + EventHub diagnostics/monitoring metrics

**🔹 Validation Objectives**

* Ensure that all messages sent to EventHub:
  + Are received and ingested into Bronze
  + Are accurately transformed and appear in Silver
* Confirm **1:1 or n:1 mapping (if business logic applies)** across layers
* Detect **record loss, duplication, or processing failure**

**🧪 Testing Approach**

**✅ Perform Count Consistency Validation**

1. Define a **controlled time window** (e.g., 15-minute or hourly load)
2. Identify count of messages sent to EventHub during this window using:
   * EventHub monitoring or metrics
   * MuleSoft logs (if applicable)
   * Test publishing script (known count)
3. Query Bronze layer for ingested message count using:
   * \_ingest\_time, event\_type, or eventhub\_offset
4. Query Silver layer for transformed record count using:
   * load\_dt, event\_id, or mapped business keys
5. Compare:
   * **EventHub count vs. Bronze** (validate ingestion)
   * **Bronze count vs. Silver** (validate transformation integrity)
6. Log and explain any differences:
   * Valid exclusions (e.g., filtered test messages)
   * Malformed or duplicate data
   * Ingestion or transformation failures

**📋 Validation Methods**

* Compare counts using:
  + Azure EventHub metrics: messages\_in\_total
  + SQL queries on Bronze and Silver:

sql

CopyEdit

SELECT COUNT(\*) FROM bronze WHERE \_ingest\_time BETWEEN X AND Y

SELECT COUNT(\*) FROM silver WHERE load\_dt = CURRENT\_DATE

* Reconcile by event\_type, uuid, or eventhub\_offset
* Optionally, automate as part of DLT pipeline expectations or validation suite

**✅ Success Criteria**

* Count of records matches across EventHub → Bronze → Silver within margin of error (0% for test cases, allowable delta for real-time lag)
* Known and explainable differences are documented (e.g., filtered out test records)
* No unexplained drop or spike in records
* Record traceability is possible via IDs or timestamps

**🧾 Exclusions**

* Deduplication logic and idempotency are covered in **E2E-03**
* Performance metrics (latency, throughput) are addressed in **E2E-02**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ03 – Completeness** | Ensures all expected records are present. |
| **RL DQ06 – Reliability** | Validates consistent record delivery and tracking. |
| **RL DQ09 – Traceability** | Supports lineage validation across layers. |

**🔐 Pre-requisites & Access Required**

* Access to:
  + EventHub telemetry/diagnostic metrics
  + Bronze and Silver table data in Databricks
* Capability to:
  + Inject or track a fixed volume of test messages
  + Execute timestamp-based and ID-based reconciliation queries
  + Identify any business logic filters applied between layers

**🔍 Test Scenario ID: E2E-02 – Latency Measurement**

**🔹 Purpose**

This test measures the **end-to-end data latency** — the total time taken for a message to travel from the source system (via EventHub) and be fully available in the Silver layer for downstream use. It helps assess the **freshness** of data, identify performance bottlenecks, and ensures the pipeline meets agreed SLAs for near-real-time or batch availability.

**🔹 Scope and Components Involved**

* **Start Point**: Message generation in source system (e.g., Athena, MQS, Mule)
* **Intermediate**:
  + Ingestion into EventHub (Azure)
  + Logging into Bronze layer
* **End Point**: Final transformation and availability in Silver layer
* **Audit Fields Required**:
  + event\_generation\_time (from source payload)
  + \_ingest\_time (Bronze system timestamp)
  + load\_dt, silver\_arrival\_ts, or DLT \_commit\_time (Silver)
* **Expected Latency Windows**:
  + Streaming: ≤ 5 minutes
  + Micro-batch: ≤ 15 minutes
  + Batch: ≤ 1 hour (or based on SLA)

**🔹 Validation Objectives**

* Measure actual latency between:
  1. **Source event time → Bronze ingestion**
  2. **Bronze ingestion → Silver availability**
  3. **Source event time → Silver availability**
* Detect processing delays at any layer and provide visibility
* Alert or flag when latency exceeds expected thresholds

**🧪 Testing Approach**

**✅ Monitor Time Gaps for Sample Records**

1. Capture messages with known event\_generation\_time or inject test messages with fixed timestamps
2. Track \_ingest\_time at Bronze and final arrival timestamps at Silver (e.g., silver\_arrival\_ts)
3. Compute:
   * bronze\_latency = \_ingest\_time - event\_generation\_time
   * silver\_latency = silver\_arrival\_ts - \_ingest\_time
   * e2e\_latency = silver\_arrival\_ts - event\_generation\_time
4. Compare computed latency against expected SLA thresholds

**📋 Validation Methods**

* Use SQL or notebook queries to:
  + Sample latency metrics across message types and hours
  + Flag outliers beyond SLA limits
* Automate with pipeline observability tools:
  + DLT event logs
  + Azure Monitor or Databricks metrics

**✅ Success Criteria**

* Median and 95th percentile latencies fall within SLA thresholds
* No consistent or unexplained lags beyond defined tolerances
* Latency is predictable across peak and off-peak periods
* Pipeline can scale to meet latency expectations under load

**🧾 Exclusions**

* Count consistency is validated in **E2E-01**
* Idempotency and replay handling are covered in **E2E-03**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ06 – Reliability** | Ensures timely availability of data for analytics/consumption |
| **RL DQ08 – Timeliness** | Measures how current or fresh the data is |
| **RL DQ09 – Traceability** | Confirms audit trail through timestamps at each stage |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Event payloads with original generation timestamp (event\_generation\_time)
  + Bronze and Silver tables with ingest and arrival timestamps
  + Pipeline logs or observability dashboards
* Ability to:
  + Inject or trace messages end-to-end
  + Monitor latency across ingestion and transformation layers

**🔍 Test Scenario ID: E2E-03 – Reprocessing Idempotency**

**🔹 Purpose**

This test ensures that **reprocessing the same data does not create duplicates** or inconsistent outcomes in the Silver layer. In modern data platforms, **idempotency** is critical to safely support restarts, retries, and backfills — especially in streaming or micro-batch pipelines like those driven by Delta Live Tables (DLT).

**🔹 Scope and Components Involved**

* **Data Flow**: EventHub → Bronze → Silver
* **Event Types**: Any (e.g., QUOTE\_REQUEST, PRECOM\_RESPONSE)
* **Fields for Idempotency Check**:
  + Unique identifiers (message\_id, event\_id, uuid)
  + Partition keys (event\_time, load\_dt)
  + Audit metadata (e.g., \_ingest\_time, \_commit\_version)
* **Transformation Logic**:
  + Should enforce 1:1 mapping between input and output
  + Must ignore or overwrite already-processed data based on keys

**🔹 Validation Objectives**

* Re-ingesting the **same payload multiple times**:
  + Should not create duplicate records in Bronze or Silver
  + Should not trigger repeated transformations or aggregations
* Validate that **natural keys or deduplication logic** is correctly enforced
* Confirm **Delta table merge/upsert behavior** (if applicable) prevents data anomalies

**🧪 Testing Approach**

**✅ Perform Reprocessing Simulation**

1. Ingest a set of known test records into EventHub (e.g., 10 messages)
2. Confirm they are processed into Bronze and transformed into Silver
3. **Replay the exact same records** into EventHub again:
   * Option 1: Resend from source (Athena/Mule)
   * Option 2: Use Azure EventHub Capture replay
4. Observe behavior:
   * Bronze: no new duplicates OR correct deduplication tags
   * Silver: no new duplicates, no change in aggregates or results
   * Logs: no errors, retries, or excessive recomputations

**📋 Validation Methods**

* Use unique message IDs or payload hashes to detect duplicates
* Compare record counts before and after reprocessing
* Use versioning or audit fields (e.g., \_commit\_version, \_last\_updated\_at) to ensure no unintended writes
* Check whether deduplication logic is enforced via:
  + ROW\_NUMBER() OVER (PARTITION BY message\_id ORDER BY \_ingest\_time DESC)
  + DLT expect\_no\_duplicates

**✅ Success Criteria**

* Re-ingested records do not increase Silver row count
* No duplicated facts or metrics appear due to reprocessing
* Bronze table may optionally log duplicates but must flag or quarantine
* System logs show idempotent processing behavior with no errors

**🧾 Exclusions**

* Message count consistency is tested in **E2E-01**
* Timestamp-based lag and SLA validation is handled in **E2E-02**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ03 – Completeness** | Prevents over-counting due to duplicates |
| **RL DQ06 – Reliability** | Confirms system behaves consistently during retries |
| **RL DQ09 – Traceability** | Ensures events can be traced uniquely and deterministically |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Bronze and Silver layer tables
  + Ingest logs or EventHub replay interface
* Ability to:
  + Replay identical payloads
  + Track message lifecycle via identifiers or timestamps
* DLT deduplication strategy or merge logic must be clearly defined and available

**🔍 Test Scenario ID: E2E-04 – Error Logging & Alerting**

**🔹 Purpose**

This scenario ensures that **any data or processing error** across the EventHub → Bronze → Silver pipeline is properly **captured, logged, and triggers alerts as needed**. The objective is to guarantee **early visibility** into ingestion or transformation issues, enabling proactive resolution and compliance with operational SLAs.

**🔹 Scope and Components Involved**

* **Pipeline Coverage**:
  + EventHub ingestion issues (malformed payloads, throttling)
  + Bronze layer write failures (schema mismatch, nulls)
  + Silver layer transformation failures (business logic exceptions)
  + Infrastructure or job execution issues (DLT job failures, cluster unavailability)
* **Monitoring Layers**:
  + Azure Monitor (for EventHub and Mule APIs)
  + Databricks log capture (DLT logs, notebook exceptions)
  + Logging destinations: ADLS logs, system tables, dashboards
* **Alert Channels**:
  + Email, Teams, PagerDuty, Azure Alerts

**🔹 Validation Objectives**

* Verify that any ingestion or processing error is:
  + Logged with sufficient metadata (timestamp, error type, impacted payload ID)
  + Accessible via queryable logs or observability dashboards
  + Linked to an alert rule when error thresholds are breached
* Confirm proper tagging/classification of:
  + Transient issues (e.g., retries succeeded)
  + Hard failures (e.g., transformation aborted)

**🧪 Testing Approach**

**✅ Simulate Controlled Failure Conditions**

1. Inject malformed or corrupt messages into EventHub (e.g., missing mandatory fields)
2. Simulate a schema evolution issue not yet handled (extra or renamed fields)
3. Cause a transformation logic exception in Silver (e.g., divide by zero, null dereference)
4. Temporarily disable a mapping table or dependency used in the Silver pipeline
5. For each scenario, verify:
   * Bronze or Silver gracefully quarantines or skips bad records
   * Logs are captured with full stack trace or root cause
   * Alerts are triggered if error rate exceeds a configured threshold (e.g., 5 failures in 10 min)

**📋 Validation Methods**

* Query DLT event logs or system error table:

sql

CopyEdit

SELECT \* FROM silver\_error\_log WHERE event\_time > CURRENT\_DATE()

* Monitor Azure EventHub diagnostic logs for ingestion errors
* Validate logs have:
  + event\_id, error\_message, failed\_stage, source\_payload, timestamp
* Observe alerting platforms (email, Slack, Teams) for real-time notifications

**✅ Success Criteria**

* 100% of errors are logged with proper classification (recoverable vs. hard failure)
* Alert is generated when error frequency breaches defined thresholds
* Logs enable root cause tracing using identifiers or metadata
* Pipeline does not fail silently — either retries, quarantines, or fails loudly with logging

**🧾 Exclusions**

* Business rule filtering (intentional record drops) is handled in **BR-SL-05**
* Malformed payload validation is tested in **EH-BR-04**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ06 – Reliability** | Confirms pipeline robustness and error recoverability |
| **RL DQ09 – Traceability** | Ensures failure points are traceable for diagnostics |
| **RL DQ10 – Observability** | Guarantees system-level transparency of issues |

**🔐 Pre-requisites & Access Required**

* Access to:
  + DLT logs, job run metadata, and Silver error quarantine tables
  + Azure Monitor and alerting dashboards
* Ability to:
  + Inject invalid data
  + Temporarily simulate dependency failures
  + View logs and test alerting configuration

**🔍 Test Scenario ID: E2E-05 – Backfill Handling**

**🔹 Purpose**

This scenario validates that the ingestion pipeline can successfully handle **backfill data loads** — historical events that are either late-arriving or replayed due to system outages, late file drops, or business reprocessing needs. The aim is to ensure that historical data can be **processed accurately without duplicates, order issues, or pipeline breaks**, while remaining traceable and conforming to the same transformation standards.

**🔹 Scope and Components Involved**

* **Backfill Use Cases**:
  + Retrospective file-based loads (e.g., legacy MQS message files)
  + Resending historical messages via EventHub (via Mule/API replays)
  + Recovery from data gaps due to ingestion outages or partition misses
* **Impact Zones**:
  + EventHub (message timestamps ≪ current time)
  + Bronze (late arrival metadata capture)
  + Silver (business rules may rely on event\_time, not load\_dt)
* **Fields for Validation**:
  + event\_time (original business timestamp)
  + \_ingest\_time (actual arrival in pipeline)
  + load\_type = backfill or replay if tagged

**🔹 Validation Objectives**

* Confirm the pipeline accepts and processes historical records as intended
* Validate backfill messages don’t overwrite or corrupt existing data
* Check that deduplication logic, windowing rules, and time-based aggregations correctly accommodate late data
* Ensure backfill records are traceable and tagged in metadata

**🧪 Testing Approach**

**✅ Execute Historical Load Simulation**

1. Prepare a controlled dataset with timestamps set to **past dates** (e.g., 1–3 months old)
2. Ingest via:
   * API replay or Mule test endpoint
   * EventHub Capture reprocessing (if available)
   * Direct Bronze injection (for file-based test)
3. Verify:
   * Bronze records are accepted and correctly tagged with \_ingest\_time, event\_time, and optionally load\_type = 'backfill'
   * Silver reflects transformed results as expected for the historical period
   * No duplication, overwriting, or record loss in previously existing data
   * Any date-based rules, e.g., validity periods, are applied accurately to old timestamps

**📋 Validation Methods**

* Track lineage using:
  + event\_id, message\_id, load\_type, and event\_time
* Query Silver records to ensure:
  + Historical partitions are updated correctly (e.g., 2023-12-01)
  + Record count matches expected from backfill file
* Monitor logs for:
  + Pipeline retries or exceptions
  + Warnings on out-of-order arrival or time skew

**✅ Success Criteria**

* Backfill records are processed without breaking the pipeline
* Silver layer reflects accurate business logic results for the old data period
* No duplication of already-processed data (if backfill overlaps with existing range)
* Metadata clearly shows which records were loaded as backfill (for traceability)

**🧾 Exclusions**

* Reprocessing duplicate detection is covered in **E2E-03**
* Timestamp normalization and ingestion SLA checks are handled in **EH-BR-03** and **E2E-02**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ03 – Completeness** | Ensures no historical gaps or missed data |
| **RL DQ09 – Traceability** | Allows identifying source/load time of older events |
| **RL DQ06 – Reliability** | Validates robust handling of replayed or backfilled data |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Bronze and Silver tables
  + Historical test datasets with original timestamps
* Ability to:
  + Inject backfill messages via Mule/EventHub
  + Identify and query backfill partitions in downstream tables
  + Monitor logs for replay-specific messages or issues

**🔍 Test Scenario ID: E2E-06 – Cross-Layer Field Value Consistency**

**🔹 Purpose**

This test ensures that key business and technical fields maintain **value consistency** as records propagate from **EventHub → Bronze → Silver**. It validates that no unintended transformations, truncations, or encoding issues alter original message values, especially for high-importance fields like event\_type, event\_time, message\_id, product\_code, and quote\_id.

**🔹 Scope and Components Involved**

* **Involved Fields**:
  + Technical identifiers: message\_id, eventhub\_offset, event\_type
  + Business fields: quote\_id, cache\_id, rating\_score, product\_code
  + Timestamps: event\_time, \_ingest\_time, silver\_arrival\_ts
* **Layers**:
  + **EventHub** (message payloads sent from source)
  + **Bronze** (raw landing of messages)
  + **Silver** (post-transformation business table)
* **Common Issues Detected**:
  + Character encoding mismatches
  + Truncation or field loss during parsing
  + Accidental overwrites in joins or transformations
  + Null propagation for optional fields

**🔹 Validation Objectives**

* Confirm that **field values are preserved accurately** through ingestion and transformation
* Ensure consistent interpretation of data types, encoding, and nullability
* Validate that **derived or mapped values** (if applicable) still reflect original semantics
* Detect inconsistencies introduced by:
  + Schema evolution
  + ETL logic errors
  + Business rule overrides

**🧪 Testing Approach**

**✅ Compare Sample Records Across Layers**

1. Ingest known test records with identifiable values:
   * E.g., quote\_id = Q123456, rating\_score = 87.25, event\_time = 2024-12-31 14:45:00
2. Trace these records through:
   * **EventHub payload log** (source message)
   * **Bronze raw table**
   * **Silver business table**
3. For each critical field, compare:
   * Raw value vs. parsed Bronze value
   * Bronze value vs. transformed Silver value
4. Validate:
   * No data loss, rounding errors, or null injection
   * Business fields maintain original precision and structure

**📋 Validation Methods**

* Use joins between Bronze and Silver using event\_id or message\_id
* Create exception tables or validation views:

sql

CopyEdit

SELECT \* FROM bronze b

JOIN silver s ON b.message\_id = s.message\_id

WHERE b.rating\_score != s.rating\_score

OR b.quote\_id != s.quote\_id

* Flag any mismatches for review and fix

**✅ Success Criteria**

* 100% of validated fields match exactly across layers
* No truncation, corruption, or null value propagation without justification
* Derived fields reflect the correct source semantics
* All mismatches are explainable (e.g., expected business transformation)

**🧾 Exclusions**

* Data type casting is addressed in **BR-SL-04**
* Field mapping coverage is detailed in **BR-SL-03**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ04 – Accuracy** | Validates correctness and precision of critical fields |
| **RL DQ06 – Reliability** | Ensures data remains intact through pipeline stages |
| **RL DQ09 – Traceability** | Supports lineage of field-level changes |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Source payload (EventHub logs or test input)
  + Bronze and Silver tables
* Ability to:
  + Inject controlled test messages
  + Compare values via notebook or automated test scripts
* Metadata or documentation of expected transformations (if any)

**🔍 Test Scenario ID: OP-01 – Pipeline Failure Recovery**

**🔹 Purpose**

This scenario ensures that the data pipeline can **recover gracefully from unexpected failures**, such as job crashes, node interruptions, or cluster timeouts, without data loss or corruption. The test validates that the system automatically resumes from the correct checkpoint and continues processing **without duplicating** or **skipping messages**.

**🔹 Scope and Components Involved**

* **Failure Types Simulated**:
  + Databricks DLT job crash or manual termination
  + Azure EventHub disconnection or throttling
  + Underlying storage (ADLS) latency or unavailability
  + Transformation logic exceptions causing job halt
* **Pipeline Layers Validated**:
  + Bronze layer: streaming ingestion
  + Silver layer: batch/micro-batch transformation
  + EventHub offset tracking and DLT checkpoints
* **Key Recovery Aspects**:
  + Checkpoint resumption
  + Offset retention
  + Deduplication during retries

**🔹 Validation Objectives**

* Confirm that pipeline resumes from the **last successful checkpoint** after failure
* Ensure **no data duplication or loss** occurs during restart
* Validate logs and metadata entries reflect proper restart state
* Verify operational alerting and logging are triggered on job failure

**🧪 Testing Approach**

**✅ Simulate Failures and Validate Recovery**

1. Start the normal EventHub → Bronze → Silver data flow
2. Inject identifiable sample records into EventHub (e.g., message\_id = TEST\_RECOVERY\_01)
3. **Force a failure** mid-ingestion:
   * Manually stop the DLT job
   * Introduce schema error or transformation exception
   * Kill compute cluster or simulate quota exceed
4. After failure, **restart the pipeline**
5. Validate:
   * Processing resumes **from correct offset/checkpoint**
   * No messages are reprocessed twice (no duplicates in Silver)
   * Expected number of records still appear in Bronze and Silver
   * Logs show checkpoint restoration (e.g., "Resuming from offset X")

**📋 Validation Methods**

* Use DLT system event logs to confirm checkpoint markers
* Query message counts and sample records before and after failure
* Check for duplicate keys (e.g., message\_id) in Silver
* Use control table or audit logs to detect skipped or duplicate ingestion

**✅ Success Criteria**

* Pipeline resumes without manual intervention or full reprocessing
* No records are lost or duplicated after recovery
* Alerts or failure logs clearly indicate root cause and restart behavior
* Data remains consistent and traceable throughout recovery

**🧾 Exclusions**

* Duplicate prevention logic is covered in **E2E-03**
* Message completeness is validated under **EH-BR-01**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ06 – Reliability** | Ensures pipeline handles failure and recovery without data loss |
| **RL DQ09 – Traceability** | Verifies recovery metadata and job event logs support auditing |
| **RL DQ10 – Observability** | Validates error alerts and health metrics during failure events |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Databricks DLT job configuration and logs
  + Azure EventHub and checkpoint metadata
  + Bronze/Silver tables for post-restart validation
* Ability to:
  + Trigger pipeline stop/failure
  + Restart job manually or verify auto-recovery behavior

**🔍 Test Scenario ID: OP-02 – Checkpoint Verification**

**🔹 Purpose**

This test ensures that the ingestion and transformation pipelines use **checkpointing mechanisms properly** to resume processing without reprocessing already-handled data. It verifies that checkpoints are **persisted, updated, and respected** across EventHub ingestion and DLT job executions, which is critical for **idempotency, latency control, and pipeline recovery**.

**🔹 Scope and Components Involved**

* **Relevant Layers**:
  + **EventHub Consumer Group Offsets** – ensures Bronze resumes from last read offset.
  + **Delta Live Tables (DLT) Checkpoints** – persists batch/job progress for Bronze → Silver.
  + **Metadata Tables** or \_checkpoint files in cloud storage (ADLS) if used.
* **Checkpoint Types**:
  + Streaming offset checkpoints for EventHub to Bronze
  + Job state checkpoints for Bronze to Silver (DLT transaction tracking)
* **Typical Issues Detected**:
  + Checkpoints not updating after success
  + Checkpoints lost or rolled back after failure
  + Partial commits causing duplicate ingestion or processing skips

**🔹 Validation Objectives**

* Confirm checkpoint state is updated **after each successful pipeline run**
* Ensure pipeline restarts **from last processed record**, not the beginning
* Validate that **no duplicate** or **missing records** are seen across re-runs
* Review **checkpoint visibility and storage** (logs, system tables)

**🧪 Testing Approach**

**✅ Simulate Controlled Runs and Restarts**

1. Start a DLT pipeline and process a known dataset (e.g., 10 events)
2. Stop the pipeline gracefully and inspect:
   * Bronze checkpoint (e.g., EventHub offset stored)
   * Silver checkpoint (job run state or version)
3. Resume the pipeline and send:
   * A **new set of messages**
   * Confirm **only new data** is processed (no replay)
4. Manually trigger a **pipeline failure** (e.g., transformation error)
5. Restart the job and validate:
   * It resumes from the **last successful state**
   * All downstream records are still **accurate and unduplicated**

**📋 Validation Methods**

* Use Databricks log views or system tables to inspect checkpoint offsets:
  + DLT event logs
  + Streaming query progress (offsets, watermarks)
* Compare record counts in Bronze and Silver **before and after restarts**
* Join message IDs across retries to verify **no replay of previously processed messages**

**✅ Success Criteria**

* Checkpoints are updated after successful batch runs
* Job resumes from exact state on restart or failure recovery
* No message duplication or missing records
* Logs clearly reflect checkpoint positions and update success

**🧾 Exclusions**

* Message replay validation logic is covered in **E2E-03**
* End-to-end pipeline fault handling is tested in **OP-01**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ06 – Reliability** | Guarantees system recovers with correct processing state |
| **RL DQ09 – Traceability** | Ensures that job state and progress are auditable |
| **RL DQ10 – Observability** | Confirms checkpoint behavior is logged and visible for monitoring |

**🔐 Pre-requisites & Access Required**

* Access to:
  + DLT pipeline configuration and log output
  + EventHub Consumer Group monitoring (Azure portal)
  + Delta \_commit\_version metadata (Silver table)
* Permissions to:
  + Start, stop, and restart ingestion jobs
  + Simulate failures and observe job restarts

**🔍 Test Scenario ID: OP-03 – Multiple Message Type Processing**

**🔹 Purpose**

This scenario validates the system's ability to process and differentiate **multiple message types** originating from the source system (e.g., MQS or Athena), which send a range of structured events such as QUOTE\_REQUEST, PRECOM\_REQUEST, PRECOM\_RESPONSE, RATING\_REQUEST, and STEP\_RESPONSE. It ensures that schema detection, branching logic, and transformation flows are correctly configured to support varied payloads without failure or misrouting.

**🔹 Scope and Components Involved**

* **Message Types Handled**:
  + QUOTE\_REQUEST
  + PRECOM\_REQUEST
  + PRECOM\_RESPONSE
  + RATING\_REQUEST
  + STEP\_RESPONSE
* **Pipeline Components**:
  + EventHub ingestion stream
  + Bronze (raw ingestion table)
  + Silver transformation logic (distinct flows per message type)
  + Routing logic (if implemented in Mule, EventHub consumer, or DLT)
* **Expected Behaviors**:
  + Schema-specific parsing and validation
  + Business rules applied conditionally by message type
  + Message-type tagging for downstream traceability

**🔹 Validation Objectives**

* Confirm all known message types are **ingested, parsed, and written** to Bronze without error
* Ensure correct **message type detection and tagging**
* Validate distinct transformation logic per message type is triggered correctly in Silver
* Ensure invalid or unknown message types are **quarantined, logged, or ignored** appropriately

**🧪 Testing Approach**

**✅ End-to-End Simulation for Each Type**

1. Send at least one sample message of **each known type** via test injection route (MQS replay, Mule, or EventHub)
2. Ingest and verify:
   * Bronze table logs the message with correct schema and \_message\_type
   * Silver table applies the **relevant transformation logic** per type (e.g., rating score mapping only for RATING\_REQUEST)
3. Intentionally send:
   * A malformed message with an **unknown or missing type**
   * A valid structure but **unexpected fields** for a known type
4. Validate:
   * Pipeline doesn’t fail globally
   * Unknown or unexpected formats are **logged or quarantined**, not dropped silently

**📋 Validation Methods**

* Query Bronze with filter on \_message\_type for each known value
* Validate record counts match the number of injected samples
* Review logs or output tables for transformation triggers per message type
* Validate that enriched fields are populated based on type-specific rules

**✅ Success Criteria**

* All expected message types are successfully parsed and processed
* Transformation logic executes correctly per type without cross-contamination
* Invalid or unknown messages are gracefully handled and traceable
* Silver outputs reflect correct branching behavior based on message type

**🧾 Exclusions**

* Schema validation and backward compatibility checks are covered under **SCHEMA-01**
* Null/corrupt message handling is tested under **EH-BR-04**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Validity** | Ensures messages conform to type-specific structures |
| **RL DQ03 – Completeness** | Confirms all types of business events are accounted for |
| **RL DQ06 – Reliability** | Validates pipeline supports schema and routing complexity |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Sample payloads for each message type
  + Ingestion audit logs, Bronze and Silver tables
* Ability to:
  + Inject test messages (Mule/test API/EventHub dev stream)
  + Query transformation logic per type

**🔍 Test Scenario ID: OP-04 – Rate Limiting / Throttling Test**

**🔹 Purpose**

This scenario validates how the pipeline responds when a **sudden burst of messages** is sent from the source system (e.g., MQS via Mule to EventHub) beyond its regular throughput. It ensures that the system remains stable, messages are **not dropped**, and **backpressure or retries** are correctly handled through EventHub buffering, Bronze streaming, and Silver processing stages.

**🔹 Scope and Components Involved**

* **Sources**:
  + MQS/MuleSoft producers generating test messages
  + EventHub partitions receiving message spikes
* **Target Components**:
  + Bronze layer (Delta Live Table streaming jobs)
  + Silver layer (batch/micro-batch logic)
* **Stress Scenarios Tested**:
  + Short-term spike (e.g., 10x message volume within 1 minute)
  + Sustained high-rate ingestion for ~30–60 mins
  + Peak-hour simulation alongside normal loads

**🔹 Validation Objectives**

* Confirm **EventHub partitions buffer data** without message loss or errors
* Validate that Bronze layer streaming logic can **handle backlog gracefully**
* Monitor for **latency spikes, processing delay, and error handling**
* Ensure Silver layer catches up once load normalizes
* Confirm **no alerts or outages** are missed during throttling conditions

**🧪 Testing Approach**

**✅ Perform Controlled Load Testing**

1. Use a load generation tool or script to simulate a **burst of messages** (e.g., 5,000–10,000 events) pushed into EventHub within a short window
2. Monitor:
   * EventHub metrics (buffer length, throughput, throttling)
   * Bronze ingestion lag and checkpoint advancement
   * Silver job progress and delays
3. Optionally:
   * Send burst per message type to test schema load handling
   * Track end-to-end latency for high-traffic test records

**📋 Validation Methods**

* Use Azure EventHub portal to monitor:
  + Incoming vs. outgoing messages
  + Partition buffer fill and throttling alerts
* Check Databricks job logs and metrics for:
  + Lag (in seconds or number of records)
  + Processing batch size, delays, and backpressure handling
* Analyze Bronze and Silver arrival times to evaluate latency during the load

**✅ Success Criteria**

* No data is lost or dropped during high load
* EventHub and DLT pipeline recover and **process entire burst**
* Lag metrics increase temporarily but **return to normal** after the spike
* Alerts are triggered appropriately (if thresholds exceeded)

**🧾 Exclusions**

* Message receipt verification is handled in **EH-BR-01**
* Latency tracking under normal load is part of **E2E-02**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ06 – Reliability** | Validates system stability under load |
| **RL DQ10 – Observability** | Confirms visibility into backlogs and throttling |
| **RL DQ03 – Completeness** | Ensures no data is lost due to ingestion pressure |

**🔐 Pre-requisites & Access Required**

* Ability to:
  + Simulate or send high-volume message loads via Mule or test EventHub stream
  + Monitor EventHub and Databricks metrics (e.g., via Azure Monitor or Ganglia)
* Access to:
  + Bronze and Silver tables for lag/latency analysis
  + Pipeline configuration to adjust throttling/resiliency settings if needed

**🔍 Test Scenario ID: OP-05 – Alerting on Message Lag**

**🔹 Purpose**

This test ensures that the pipeline monitors and raises **real-time alerts** when there is a **lag between message generation and processing**. Message lag might occur due to issues like throttling, pipeline delays, system failures, or high load. It is critical to detect such lags early to prevent data freshness issues, reporting inaccuracies, or SLA breaches.

**🔹 Scope and Components Involved**

* **Source Layer**:
  + Azure EventHub (where message lag originates)
* **Processing Layers**:
  + Bronze (streaming ingestion via DLT)
  + Silver (business-ready transformation)
* **Monitoring Systems**:
  + Azure Monitor, Databricks metrics, custom log-based alerts
* **Lag Points to Track**:
  + **EventHub enqueued time** vs. **Bronze \_ingest\_time**
  + **\_ingest\_time** vs. **Silver processing timestamp**

**🔹 Validation Objectives**

* Confirm that lag thresholds are **defined and configurable** (e.g., 5 minutes, 15 minutes)
* Ensure **alerts are triggered** when lag crosses defined limits
* Validate alerts are routed via appropriate channels (email, dashboard, log entry)
* Ensure lag **resolves automatically** when processing resumes
* Confirm no false positives or alert storms under normal conditions

**🧪 Testing Approach**

**✅ Simulate Controlled Lag and Monitor Response**

1. Send test messages to EventHub with known timestamps
2. Introduce artificial lag:
   * Pause Bronze ingestion job temporarily
   * Delay Silver processing job or simulate failure
3. Monitor:
   * Delay between EventHub enqueue time and Bronze/Silver arrival
   * Alert generation through Azure Monitor, Databricks logs, or notification channels
4. Resume pipeline and verify:
   * Lag is reduced and alert state is cleared
   * All messages are still processed correctly

**📋 Validation Methods**

* Compare eventhub\_enqueued\_time with \_ingest\_time and Silver arrival timestamp
* Monitor alert dashboards or logs
* Review automated alert records (e.g., alerts fired in last 24h)
* Analyze how long lag persisted and how quickly system responded

**✅ Success Criteria**

* Alerts trigger **within threshold breach window**
* Alert messages are accurate and informative (include lag duration, pipeline ID)
* System recovers without manual intervention once lag condition clears
* Historical logs reflect alert state transitions (raised, acknowledged, cleared)

**🧾 Exclusions**

* Normal latency monitoring is covered in **E2E-02**
* Recovery behavior is covered in **OP-01**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ10 – Observability** | Ensures visibility into message delay and ingestion health |
| **RL DQ06 – Reliability** | Validates pipeline alerts when health degrades |
| **RL DQ08 – Timeliness** | Ensures data availability stays within freshness SLAs |

**🔐 Pre-requisites & Access Required**

* Ability to:
  + Modify or pause DLT job temporarily to simulate lag
  + Send timestamped messages to EventHub
* Access to:
  + Monitoring dashboards (Azure Monitor, Databricks metrics)
  + Notification configurations (email, Slack, Teams, etc.)

**🔍 Test Scenario ID: OP-06 – Ingestion Job Scheduling and Trigger Verification**

**🔹 Purpose**

This scenario ensures that the data ingestion and transformation jobs (DLT pipelines or notebooks) are triggered **at the correct frequency or based on event-driven schedules** as defined in the data platform architecture. It verifies **no missed or overlapping job executions**, which can lead to data delay, partial loads, or job conflicts.

**🔹 Scope and Components Involved**

* **Ingestion Types**:
  + **Streaming** (continuous Bronze jobs)
  + **Batch/Micro-batch** (Silver transformation jobs, often scheduled)
* **Trigger Methods**:
  + Time-based (e.g., every 15 min/hourly)
  + Event-based (e.g., new data arrival triggers processing)
* **Infrastructure Components**:
  + Databricks Workflows (job orchestration)
  + DLT Scheduler / Job cluster logs
  + External schedulers if used (e.g., Airflow, Azure Data Factory)

**🔹 Validation Objectives**

* Confirm job scheduling is aligned with SLA and business requirements
* Validate job actually starts at defined schedule or trigger
* Ensure **no overlapping executions** or missing jobs
* Check dependencies (e.g., Bronze completion before Silver starts)
* Ensure trigger failures are logged and alerted

**🧪 Testing Approach**

**✅ Schedule Verification Steps**

1. Identify job frequency from scheduling config or orchestration pipeline
2. Monitor:
   * Job execution history (Databricks job UI or system audit table)
   * Job start and end times over a 24-hour period
   * Dependencies (e.g., Bronze job finishes before Silver starts)
3. Simulate a missed trigger:
   * Pause a dependent job to test orchestration failure
   * Introduce delay in upstream job
4. Validate:
   * Error is captured and alerted
   * Retry or manual trigger behavior is handled as per fallback config

**📋 Validation Methods**

* Review execution logs and trigger timestamps from:
  + job\_run\_metadata table or audit log tables
  + Databricks Workflows history
* Confirm execution count matches schedule expectations
* Validate no missed or duplicated runs
* Confirm Silver job respects Bronze job state (no early start)

**✅ Success Criteria**

* Job executes exactly at configured schedule or trigger
* No overlapping or skipped jobs unless expected
* Errors in job triggering are logged and optionally alerted
* End-to-end flow respects dependency hierarchy

**🧾 Exclusions**

* Data quality of actual ingested messages is covered in **EH-BR** and **BR-SL** tests
* Recovery from job failure is addressed in **OP-01**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ06 – Reliability** | Ensures job flows run predictably and consistently |
| **RL DQ10 – Observability** | Confirms job health and schedule tracking is visible |
| **RL DQ08 – Timeliness** | Verifies data is refreshed according to business frequency SLAs |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Databricks Workflows / Jobs scheduling interface
  + System audit log tables or job metadata tracking logs
* Permissions to:
  + Trigger, pause, or re-run ingestion and transformation jobs
  + Simulate or modify scheduling configuration (in test environment)

**🔍 Test Scenario ID: SCHEMA-01 – Backward-Compatible Schema Handling**

**🔹 Purpose**

This test validates that the ingestion and transformation pipeline supports **schema evolution**, particularly **backward-compatible changes** such as new optional fields or reordered fields in the EventHub messages. It ensures that the Bronze and Silver layers can handle such changes **gracefully**, without causing job failures or skipping data.

**🔹 Scope and Components Involved**

* **Schema Change Types Validated**:
  + Addition of optional fields (e.g., discount\_code, client\_reference)
  + Reordering of fields
  + Changes to default values (nullable fields)
* **Pipeline Stages Affected**:
  + EventHub ingestion to Bronze (JSON deserialization logic)
  + Bronze to Silver transformation logic (field mappings, type casting)
* **Schema Handling Strategy**:
  + Use of schema inference or explicitly defined schema (e.g., via Spark StructType)
  + Support for soft-typed fields or try\_cast in transformations

**🔹 Validation Objectives**

* Ensure that new schema versions are **accepted** by the Bronze ingestion layer
* Validate that Bronze stores all fields (old + new) where applicable
* Ensure Silver logic can operate correctly **even if new fields are absent**
* Confirm no transformation or parsing errors when processing evolving schema

**🧪 Testing Approach**

**✅ Inject Schema Changes into the Pipeline**

1. Send baseline test message with **original schema**
2. In next batch, send message with:
   * One or more **new optional fields**
   * Same business logic, but with **fields reordered**
3. Monitor ingestion pipeline:
   * Confirm Bronze table accepts both versions
   * Silver logic parses messages and applies transformations without error
4. Confirm that absence of new fields does **not cause null-pointer or logic failures**

**📋 Validation Methods**

* Review schema snapshots in Bronze for both old and new messages
* Inspect \_rescued\_data or \_corrupt\_record fields (if used)
* Validate Spark or DLT job logs for schema inference behavior
* Run row count comparisons and transformation accuracy checks

**✅ Success Criteria**

* Bronze ingests messages with new/old schema without failure
* Silver logic dynamically handles field presence/absence
* No ingestion or transformation job crashes
* Optional new fields appear where present; are null where missing

**🧾 Exclusions**

* Unknown field logging is covered in **SCHEMA-03**
* Schema registry tracking is separately addressed in **SCHEMA-02**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Validity** | Ensures schema conforms even during change |
| **RL DQ04 – Flexibility** | Supports evolution without manual intervention |
| **RL DQ06 – Reliability** | Ensures pipeline resilience to change |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Message sample generator or schema simulator (for testing schema changes)
  + Bronze and Silver table schema definitions
  + Spark/DLT logs for schema errors or rescue records
* Ability to:
  + Modify incoming message schema format (e.g., from MQS or EventHub test topic)
  + Deploy changes to transformation logic safely (test environment)

**🔍 Test Scenario ID: SCHEMA-02 – Schema Registry Integration**

**🔹 Purpose**

This scenario verifies that the ingestion pipeline references a **central schema registry** (e.g., Confluent Schema Registry, Azure Schema Registry, or internal metadata catalog) to **validate incoming messages** against approved schema versions. It ensures only compliant message formats are processed and that schema changes are versioned, documented, and validated to prevent pipeline failures.

**🔹 Scope and Components Involved**

* **Schema Registry System**:
  + May include Azure Schema Registry, Data Catalog, or Databricks-maintained version table
* **Schema-Dependent Layers**:
  + EventHub to Bronze (schema validation on ingestion)
  + Bronze to Silver (field mapping and transformations based on schema versions)
* **Control Objectives**:
  + Avoid unintended schema drift
  + Enforce known schema versions
  + Provide traceability of schema evolution

**🔹 Validation Objectives**

* Confirm the pipeline **fetches or refers to schema versions** during ingestion or transformation
* Validate that **unregistered or invalid schemas are rejected** or quarantined
* Check that **schema version is traceable** for each processed record
* Ensure schema updates go through **approval and publishing workflows**

**🧪 Testing Approach**

**✅ Simulate Known vs. Unknown Schema Use**

1. Register an approved schema version in the registry
2. Ingest messages:
   * With a matching schema → should pass
   * With an unregistered or altered schema → should be rejected or logged
3. Monitor behavior:
   * How ingestion pipeline handles version mismatch
   * If version information is tagged in Bronze (e.g., schema\_version)
   * How transformation logic behaves if schema mismatch occurs

**📋 Validation Methods**

* Check Bronze metadata or lineage fields for schema version tags
* Monitor schema validation logs or rejection queues
* Audit schema registry for schema version definitions and timestamps
* Confirm pipeline only processes messages that match registered schema

**✅ Success Criteria**

* Only messages conforming to approved schema versions are processed
* Schema version info is available for audit or traceability
* Version mismatches do not cause pipeline failure but trigger quarantine/logging
* Schema evolution is controlled and transparent

**🧾 Exclusions**

* Handling of optional or reordered fields is tested in **SCHEMA-01**
* Unknown or extra field logging is handled in **SCHEMA-03**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Validity** | Ensures data matches registered schema definitions |
| **RL DQ09 – Traceability** | Provides lineage of schema usage over time |
| **RL DQ04 – Flexibility & Control** | Ensures schema changes are authorized and tracked |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Schema Registry (Confluent, Azure, or custom metadata store)
  + Bronze ingestion logs and Silver field mappings
* Ability to:
  + Register, update, and query schema versions
  + Inject messages with known and unknown schemas for testing

**🔍 Test Scenario ID: SCHEMA-03 – Unknown Field Ignoring/Logging**

**🔹 Purpose**

This test verifies the system’s ability to process EventHub messages that may include **unexpected or unknown fields** not currently defined in the expected schema. Instead of rejecting the entire message or causing ingestion errors, the pipeline should either **log, ignore, or isolate** these fields, ensuring continuity in data flow and traceability of schema drift.

**🔹 Scope and Components Involved**

* **Message Format**: JSON payloads with unexpected key-value pairs
* **Pipeline Components**:
  + EventHub consumer (streaming logic)
  + Bronze ingestion table (Delta Live Table schema definition)
  + Silver transformation layer (expected fields only)
* **Fallback Fields**:
  + \_rescued\_data, \_corrupt\_record, or custom extra\_fields column to capture unknowns

**🔹 Validation Objectives**

* Ensure messages with unknown fields **do not fail ingestion**
* Validate unknown fields are **logged, rescued, or ignored** based on config
* Confirm no pipeline-level crashes due to minor schema mismatch
* Ensure field drift is traceable for future schema evolution analysis

**🧪 Testing Approach**

**✅ Inject Test Messages with Extra Fields**

1. Send a message with one or more **extra/unexpected fields** not defined in the schema (e.g., unused\_flag, beta\_test\_id)
2. Ingestion to Bronze should succeed
3. Check if:
   * Extra fields appear in a **rescued data column**
   * They are silently ignored with log entry
   * Any alerts or metadata indicators are generated
4. Validate that downstream transformations (Silver layer) do not fail when such messages are present

**📋 Validation Methods**

* Query \_rescued\_data column in Bronze to inspect captured unknown fields
* Review DLT ingestion logs for field drift detection messages
* Validate record counts remain consistent before and after test
* Analyze log alerts (if configured) for schema mismatch events

**✅ Success Criteria**

* Messages with extra fields are ingested successfully
* Unexpected fields are handled without pipeline interruption
* Unknown field presence is captured for auditing or schema evolution
* No data corruption or message rejection occurs unless by design

**🧾 Exclusions**

* Backward-compatible field handling is addressed in **SCHEMA-01**
* Schema version control and validation is covered in **SCHEMA-02**

**🗂 Mapped Data Quality Requirements**

| **DQ Ref** | **Description** |
| --- | --- |
| **RL DQ01 – Validity** | Ensures valid messages aren’t rejected for unexpected extras |
| **RL DQ09 – Traceability** | Captures unknown fields for analysis |
| **RL DQ04 – Flexibility** | Supports evolving structures in a resilient manner |

**🔐 Pre-requisites & Access Required**

* Access to:
  + Bronze DLT table with \_rescued\_data enabled
  + EventHub ingestion topic for message injection
  + Databricks job logs for schema handling
* Ability to:
  + Send custom test messages with unexpected fields
  + View logs or metadata capturing schema drift