DATA QUALITY & TRUST BREAKS

Real Interview Scenarios & How to Handle Them

A practical guide for Data Engineers to answer
real-world data quality and trust breaks questions with confidence

By Ankita Gulati

Senior Data Engineer | Interview Mentor

Interview Edition • Practical • Real Scenarios

Table Of Content

Scenario 1

Missing Records in a Daily ETL Job

Problem Statement

A daily ETL job **completes successfully,** but **5–10% of records are missing in the warehouse**. Downstream reporting is critical, the **2-hour SLA** is tight, and the root cause is not immediately clear.

Key Details

- ETL job completes without failure
- 5–10% records missing in target
- Downstream reporting impacted
- Root cause unknown
- SLA: 2 hours

Expected vs Actual Behavior

Expected	Actual
All source records loaded	Partial data loaded
Data completeness guaranteed	Missing records
Reports reflect full data	Reports incorrect
SLA met with confidence	SLA met but data wrong

This is a **silent data quality failure,** not a job execution failure.

Why This Problem Is Dangerous

Because:

- The job does not fail
- Monitoring shows “success”
- Stakeholders assume data is correct

Teams often react by:

- Re-running the job blindly
- Informing business too early
- Ignoring small discrepancies

But **successful execution does not guarantee correct data.**

Clarifying Questions

Before acting, a senior engineer asks:

- Are missing records consistent across runs?
- Are specific dates, partitions, or keys missing?
- Did all source partitions arrive on time?
- Were partial writes or overwrites used?
- Did upstream data arrive late?

These questions focus on **data correctness,** not job runtime.

Confirmed Facts & Assumptions

After investigation:

- Job completed without errors
- Some partitions have lower record counts
- Source data volume is higher than target
- Issue is reproducible
- Re-running may overwrite the same bad state

Interpretation:

This is a **data completeness issue**, not a compute issue.

What the System Assumes vs Reality

System Assumption	Reality
Job success = data correct	Job success ≠ data complete
All partitions processed	Some partitions missing
Re-run will fix data	Re-run may repeat issue
Monitoring caught issues	Data checks missing

ETL pipelines must validate **outcomes**, not just execution.

Root Cause Analysis

Step 1: Compare Source and Target Counts

Observed:

- Source record count > target record count
- Discrepancy isolated to specific partitions

Conclusion:
Data loss occurred during ingestion or transformation.

Step 2: Identify Missing Partitions

Observed:

- Certain partitions incomplete or skipped
- Possible late-arriving data or partial failures

This narrows the problem to **where data went missing**.

Step 3: Conceptual Root Cause

The root cause is **lack of data completeness validation:**

- Missing partitions not detected
- Job marked successful
- Incorrect data propagated downstream

This is a **data quality and validation gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Blindly re-run the job
- Ignore small percentage gaps
- Notify business without diagnosis

Right Approach

- Compare source and target counts
- Identify missing partitions or keys
- Fix root cause before re-running

Senior engineers validate **data correctness before recovery actions.**

Step 5 : Validation of Root Cause

To confirm:

- Run source vs target reconciliation
- Identify exact missing partitions
- Backfill only missing data

Outcome:
Data completeness restored without unnecessary reprocessing.

Step 6 :Corrective Actions

- Add source-to-target count checks
- Validate partition-level completeness
- Fail job if thresholds breached
- Implement automated reconciliation
- Alert on data quality, not just job status

These steps prevent silent data corruption.

Step 7 : Result After Fix

Before Fix	After Fix
Missing records	Complete data
Silent failure	Early detection
Broken reports	Accurate reports
Reactive fixes	Proactive validation

Final Resolution

- **Root Cause:** Missing data not detected due to lack of validation
- **Fix Applied:** Source–target reconciliation and targeted backfill

Key Learnings

- Job success ≠ data correctness
- Data validation is a first-class requirement
- Re-running jobs blindly is risky
- Partition-level checks prevent silent failures

Core Principle Reinforced

Always validate data completeness before declaring an ETL job successful.

Scenario 2

Duplicate Records Introduced After a Join

Problem Statement

A daily ETL job performs a join between two large tables, but downstream dashboards show **record counts inflated by nearly 2×.** The job completes successfully, yet **business metrics are incorrect,** and the **1-hour SLA** is under pressure.

Key Details

- Join between two large tables
- Downstream counts inflated ~2×
- Data volume: ~500 GB
- Business metrics impacted
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
One-to-one or controlled join	One-to-many explosion
Accurate record counts	Duplicated records
Reliable business metrics	Inflated metrics
SLA met with confidence	SLA met but data wrong

This is a **logical data correctness failure**, not an execution failure.

Why This Problem Is Dangerous

Because:

- The job completes without errors
- Performance looks normal
- Duplication is not immediately obvious

Teams often react by:

- Dropping duplicates downstream
- Re-running the job
- Ignoring minor discrepancies

But **masking duplication hides the real issue and creates long-term inconsistency.**

Clarifying Questions

Before acting, a senior engineer asks:

- Are join keys truly unique on both sides?
- Has data cardinality changed upstream?
- Is the join type correct (INNER / LEFT / FULL)?
- Are late or duplicate ingestions involved?
- Did the join create a one-to-many relationship unintentionally?

These questions focus on **data relationships,** not runtime.

Confirmed Facts & Assumptions

After investigation:

- One join key maps to multiple rows on one side
- Join logic assumes uniqueness that does not exist
- Data volume doubles post-join
- Re-running produces the same inflation
- Dropping duplicates hides missing relationships

Interpretation:
This is a **join cardinality and logic issue.**

What the System Assumes vs Reality

Assumption	**Reality**
Join keys are unique	Join keys are not unique
Join preserves row count	Join multiplies rows
Deduplication is safe	Dedup hides logic errors
Job success means correctness	Logic errors pass silently

ETL joins must be validated for **cardinality**, not just syntax.

Root Cause Analysis

Step 1: Analyze Join Cardinality

Observed:

- Row count doubles after join
- Multiple matches per join key

Conclusion:
Join is unintentionally one-to-many.

Step 2: Inspect Join Conditions

Observed:

- Missing join predicates
- Incorrect or incomplete join keys
- Possible duplicate ingestion upstream

This confirms **faulty join logic.**

Step 3: Conceptual Root Cause

The root cause is **incorrect join assumptions:**

- Join keys not unique
- Missing constraints or filters

- Data duplication amplified during join

This is a **data modeling and logic flaw,** not a processing issue.

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Drop duplicates in target
- Restart job

Right Approach

- Fix join keys and join conditions
- Enforce uniqueness where required

Senior engineers fix **logic at the source**, not symptoms downstream.

Step 5 : Validation of Root Cause

To confirm:

- Test join cardinality before and After Fix
- Validate row counts at each stage
- Compare metrics with Expected values

Outcome:

Counts stabilize and metrics align with business expectations.

Step 6 :Corrective Actions

- Validate join key uniqueness
- Fix join predicates and filters
- Add pre-join deduplication if required
- Add row-count assertions after joins
- Document Expected join cardinality

These steps prevent silent data inflation.

Step 7 : Result After Fix

Before Fix	**After Fix**
Inflated counts	Accurate counts
Hidden logic errors	Correct join logic
Broken metrics	Trusted metrics
Temporary patches	Durable solution

Final Resolution

- **Root Cause:** Incorrect join keys or logic causing row multiplication
- **Fix Applied:** Corrected join conditions and enforced cardinality

Key Learnings

- Joins are the #1 source of silent data bugs
- Cardinality matters more than syntax
- Deduplication is not a fix for bad joins
- Validate row counts after every join

Core Principle Reinforced

Fix join logic at the source—never hide duplication with downstream deduplication.

Scenario 3

Data Type Mismatch Breaks the ETL Pipeline

Problem Statement

A daily ETL job **fails unExpectedly** because an upstream system changed a column's data type from **INT → STRING.** Multiple downstream tables depend on this column, rollback is not possible, and the **1-hour SLA** is at risk.

Key Details

- Upstream schema change: INT → STRING
- ETL job fails at runtime
- Multiple downstream dependencies
- Rollback not possible
- SLA: 1 hour

Expected vs Actual Behavior

Expected	**Actual**
ETL handles minor schema drift	Job fails on type mismatch
Pipeline remains stable	Downstream tables blocked
SLA protected	SLA breached
Schema change detected early	Failure detected at runtime

This is a **schema evolution failure,** not a compute or data volume issue.

Why This Problem Is Common

Because:

- Upstream teams evolve schemas independently
- Type changes are often considered “minor”
- ETL pipelines assume fixed schemas

Many pipelines break not due to logic errors, but due to **unhandled schema drift.**

Clarifying Questions

Before acting, a senior engineer asks:

- Which column changed type?
- Is the change backward compatible?
- Are downstream tables expecting numeric or string semantics?
- Can the column be safely cast?
- Should invalid values be quarantined?

These questions prioritize **data continuity over perfection.**

Confirmed Facts & Assumptions

After investigation:

- Column type changed from INT to STRING upstream
- Values are still numerically representable
- ETL fails during parsing or transformation
- Downstream tables are blocked
- Waiting for upstream rollback is not an option

Interpretation:

This is a **type-handling gap in the ETL logic**.

What the Pipeline Assumes vs Reality

Assumption	Reality
Schema is stable	Schema evolves
Types never change	Types drift
ETL can fail fast	Business cannot wait
Upstream will rollback	Rollback unavailable

ETL systems must be designed for **schema evolution.**

Root Cause Analysis

Step 1: Identify the Failing Transformation

Observed:

- Failure during read or cast
- Type mismatch exception thrown

Conclusion:
ETL cannot handle dynamic type changes.

Step 2: Assess Safe Recovery Options

Options considered:

- Skipping the column (data loss)
- Casting dynamically (safe and reversible)
- Blocking pipeline (SLA breach)

Correct choice:
Cast the column appropriately.

Step 3: Conceptual Root Cause

The root cause is **rigid schema enforcement:**

- ETL assumes fixed types
- No validation or casting logic
- Upstream schema drift breaks pipeline

This is a **schema evolution design flaw.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Skip the column
- Ignore failures
- Wait only on upstream teams

Right Approach

- Cast column safely
- Validate values
- Continue pipeline execution

Senior engineers favor **graceful degradation** over hard failures.

Step 5 : Validation of Root Cause

To confirm:

- Implement safe casting
- Rerun the job
- Validate downstream tables

Outcome:
Pipeline resumes and downstream dependencies unblock.

Step 6 :Corrective Actions

- Implement explicit type casting
- Add schema validation checks
- Log and quarantine invalid values
- Monitor schema changes upstream
- Document schema evolution expectations

These steps make pipelines resilient to change.

Step 7 : Result After Fix

Before Fix	**After Fix**
ETL job fails	ETL succeeds
Downstream blocked	Downstream unblocked
SLA breached	SLA met
Rigid pipeline	Flexible pipeline

Final Resolution

- **Root Cause:** Unhandled data type drift in upstream schema
- **Fix Applied:** Safe casting and schema validation

Key Learnings

- Schema drift is inevitable
- Type safety must be balanced with resilience
- Casting is often safer than skipping
- ETL pipelines must evolve with data

Core Principle Reinforced

ETL pipelines should expect schema drift and handle it gracefully—not break on it.

Scenario 4

UnExpected Null Values in Critical Columns

Problem Statement

A daily ETL job completes successfully, but **business-critical columns contain unExpected NULL values.** Downstream dashboards rely heavily on these fields, and the **1-hour SLA** leaves little room for guesswork. The dataset is large, making blind fixes risky.

Key Details

- ETL job succeeds technically
- NULLs appear in critical columns
- Dashboards depend on these fields
- Large data volume
- SLA: 1 hour

Expected vs Actual Behavior

Expected	**Actual**
Critical columns always populated	NULL values appear
Dashboards show reliable metrics	Dashboards misleading
Data quality guaranteed	Silent data corruption
SLA met with confidence	SLA met but data wrong

This is a **data correctness issue**, not a job execution issue.

Why This Problem Is Dangerous

Because:

- The pipeline reports success
- NULLs propagate silently
- Dashboards may still render

Teams often rush to:

- Impute missing values
- Re-run the job
- Ignore small NULL percentages

But **imputing before understanding the cause can permanently corrupt business logic.**

Clarifying Questions

Before acting, a senior engineer asks:

- Which columns contain NULLs?
- Are NULLs consistent across partitions?
- Did a LEFT JOIN introduce missing values?
- Did upstream data arrive incomplete?
- Are filters removing rows unExpectedly?

These questions aim to **locate the origin of NULLs**, not mask them.

Confirmed Facts & Assumptions

After investigation:

- NULLs appear only after a join
- Source tables contain valid values
- Join keys don't match for some records
- Re-running produces the same NULLs
- Imputation would hide the join issue

Interpretation:
This is a **source or join-logic issue**, not missing data per se.

What the Pipeline Assumes vs Reality

Assumption	**Reality**
Joins preserve critical columns	Joins introduce NULLs
NULLs mean missing data	NULLs mean logic failure
Imputation is safe	Imputation hides errors
Job success means correctness	Data quality unchecked

ETL pipelines must treat NULLs in critical fields as **failures**, not values.

Root Cause Analysis

Step 1: Identify Where NULLs Appear

Observed:

- Columns populated before join
- NULLs introduced post-join

Conclusion:
Join conditions are dropping or mismatching records.

Step 2: Trace Source and Join Logic

Observed:

- Incomplete join predicates
- LEFT JOIN where INNER JOIN Expected
- Upstream keys missing or malformed

This confirms **logic-driven NULL injection.**

Step 3: Conceptual Root Cause

The root cause is **unvalidated joins or incomplete source data:**

- Join mismatch introduces NULLs
- No validation step catches it
- Data silently degrades

This is a **data quality validation gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Impute missing values
- Ignore NULLs
- Blindly re-run job

Right Approach

- Trace NULLs to source or join logic
- Fix upstream data or join conditions
- Validate critical columns explicitly

Senior engineers **trace before transforming.**

Step 5 : Validation of Root Cause

To confirm:

- Check source tables for missing keys
- Fix join logic
- Re-run only affected partitions

Outcome:
Critical columns populate correctly and dashboards recover.

Step 6 :Corrective Actions

- Add NOT NULL checks for critical columns
- Validate joins explicitly
- Fail jobs when NULL thresholds exceed limits
- Log and quarantine bad records
- Add data quality alerts

These steps prevent silent corruption.

Step 7 : Result After Fix

Before Fix	After Fix
NULLs in key columns	Complete data
Misleading dashboards	Trusted metrics
Reactive patching	Root-cause fix
SLA risk	SLA protected

Final Resolution

- **Root Cause:** Join or source data issue introducing NULLs
- **Fix Applied:** Traced and corrected upstream data / join logic

Key Learnings

- NULLs in critical fields are red flags
- Imputation is not always a fix
- Joins are a common NULL source
- Data validation must be explicit

Core Principle Reinforced

Never impute critical NULLs blindly—trace the source Before Fixing the symptom.

Scenario 5

Out-of-Range Values in Business Metrics

Problem Statement

A daily sales ETL pipeline completes successfully, but the warehouse contains **negative revenue values**, which are **logically invalid.** Business dashboards rely on this data, the pipeline runs automatically, and the **2-hour SLA** leaves limited time to respond.

Key Details

- Sales metrics include negative revenue
- ETL job completes without errors
- Dashboards depend on this data
- Pipeline runs automatically every day
- SLA: 2 hours

Expected vs Actual Behavior

Expected	Actual
Revenue ≥ 0	Negative revenue values
Business logic preserved	Business logic violated
Trusted dashboards	Misleading metrics
SLA met with confidence	SLA met but data wrong

This is a **business-rule violation,** not a technical pipeline failure.

Why This Problem Is Dangerous

Because:

- The ETL job reports success
- Dashboards still load
- Invalid values look like “real data”

Teams often react by:

- Filtering out negative values
- Hiding them in dashboards
- Not questioning upstream logic

But **filtering hides systemic calculation errors and erodes data trust.**

Clarifying Questions

Before acting, a senior engineer asks:

- Which calculation produces revenue?
- Are refunds, cancellations, or adjustments handled correctly?
- Did upstream logic change recently?
- Are negative values valid in any edge case?
- Where is the first point these values appear?

These questions focus on **business correctness,** not surface cleanup.

Confirmed Facts & Assumptions

After investigation:

- Negative values originate upstream
- Calculation logic double-applies discounts/refunds
- ETL faithfully loads incorrect results
- Filtering downstream would hide the error
- Re-running does not fix the issue

Interpretation:

This is a **source calculation error**, not an ETL transformation bug.

What the Pipeline Assumes vs Reality

Assumption	Reality
Source metrics are valid	Source logic is flawed
ETL only transforms	ETL amplifies bad logic
Filtering is safe	Filtering hides defects
Job success = data correctness	Business rules violated

ETL pipelines must enforce **business constraints,** not just move data.

Root Cause Analysis

Step 1: Trace Where Invalid Values Appear

Observed:

- Revenue values become negative before load
- ETL logic does not introduce negativity

Conclusion:
Error originates in source calculations.

Step 2: Validate Business Logic

Observed:

- Refunds or discounts applied incorrectly
- Edge cases not handled
- No lower-bound validation

This confirms **broken business logic upstream.**

Step 3: Conceptual Root Cause

The root cause is **missing business-rule validation:**

- Revenue allowed to go below zero
- ETL loads values without sanity checks
- Invalid metrics propagate to dashboards

This is a **data governance issue,** not a technical failure.

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Filter negative values
- Ignore small percentages
- Notify business without diagnosis

Right Approach

- Investigate source calculations
- Fix business logic
- Enforce valid ranges

Senior engineers fix **meaning**, not appearance.

Step 5 : Validation of Root Cause

To confirm:

- Fix upstream calculation logic
- Reprocess affected data
- Verify revenue values are non-negative

Outcome:
Metrics align with real business behavior.

Step 6 :Corrective Actions

- Enforce non-negative constraints
- Add business-rule validations in ETL
- Alert on out-of-range values
- Document metric definitions clearly
- Add automated sanity checks

These steps protect metric integrity long-term.

Step 7 : Result After Fix

Before Fix	**After Fix**
Negative revenue	Valid metrics
Misleading dashboards	Trusted dashboards
Temporary patches	Root-cause fix
Data trust erosion	Data trust restored

Final Resolution

- **Root Cause:** Incorrect upstream calculation logic
- **Fix Applied:** Corrected source calculations and added validation

Key Learnings

- Metrics must obey business rules
- ETL correctness ≠ business correctness
- Filtering hides real problems
- Validations belong in pipelines

Core Principle Reinforced

When metrics look wrong, fix the logic—never just hide the values.

Scenario 6

Slowly Drifting Metrics Over Time

Problem Statement

Daily business metrics remain within acceptable bounds initially but **gradually drift away from Expected ranges over several weeks**. Reporting continues to meet the daily SLA, but **historical trends start to look suspicious,** even though the **data source has not changed.**

Key Details

- Metrics drift slowly over weeks
- Daily reporting SLA met
- Historical trends critical for business decisions
- No obvious source or schema change
- SLA: daily reporting

Expected vs Actual Behavior

Expected	**Actual**
Metrics remain stable over time	Metrics drift gradually
Trends reflect real business changes	Trends slowly distort
Daily reports trustworthy	Long-term trust erodes
SLA met with confidence	SLA met but accuracy questionable

This is a **long-term data correctness issue**, not a daily job failure.

Why This Problem Is Dangerous

Because:

- Daily checks pass
- No sudden spikes or failures occur
- Drift looks “normal” at first

Teams often:

- Ignore small deviations
- Assume business behavior changed
- Notice the issue only after weeks

But **slow drift is harder to detect and more damaging than sudden failures.**

Clarifying Questions

Before acting, a senior engineer asks:

- When did the drift begin?
- Is the deviation linear or compounding?
- Are aggregations cumulative or incremental?
- Were default values or rounding logic introduced?
- Do recalculations match historical expectations?

These questions focus on **trend integrity,** not single-day correctness.

Confirmed Facts & Assumptions

After investigation:

- Drift increases gradually each day
- Raw source data appears unchanged
- Incremental aggregations compound small errors
- No validation against historical baselines exists
- Restarting the job does not reset drift

Interpretation:
This is a **cumulative aggregation or logic drift issue**.

What the Pipeline Assumes vs Reality

Assumption	Reality
Small errors don't matter	Errors compound over time
Daily accuracy is sufficient	Historical accuracy matters
Incremental logic is stable	Incremental logic drifts
Monitoring catches issues	Drift remains invisible

ETL pipelines must validate **trends**, not just daily snapshots.

Root Cause Analysis

Step 1: Compare Against Historical Baseline

Observed:

- Recalculated historical metrics differ from incremental results
- Gap widens over time

Conclusion:
Incremental logic is accumulating error.

Step 2: Inspect Aggregation Logic

Observed:

- Rolling averages or cumulative sums
- Rounding or default values applied repeatedly
- Missing corrections for late-arriving data

This confirms **aggregation drift.**

Step 3: Conceptual Root Cause

The root cause is **unchecked incremental aggregation logic:**

- Small daily inaccuracies
- No reconciliation with historical baseline
- Drift compounds silently

This is a **data correctness and validation gap,** not a runtime issue.

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore small deviations
- Restart ETL jobs
- Notify stakeholders without diagnosis

Right Approach

- Recalculate metrics using historical baseline
- Identify divergence point
- Fix aggregation logic

Senior engineers validate **long-term correctness**, not just daily success.

Step 5 : Validation of Root Cause

To confirm:

- Recompute metrics from raw historical data
- Compare with incremental outputs
- Identify when divergence started

Outcome:
Root cause isolated and corrected.

Step 6 :Corrective Actions

- Recalculate metrics from historical baseline
- Fix incremental aggregation logic
- Periodically reconcile incremental vs full recompute
- Add drift detection alerts
- Document metric calculation assumptions

These steps prevent silent trend corruption.

Step 7 : Result After Fix

Before Fix	After Fix
Gradual metric drift	Stable trends
Hidden inaccuracies	Verified correctness
Eroding trust	Restored confidence
Reactive investigation	Proactive monitoring

Final Resolution

- **Root Cause:** Cumulative error in incremental aggregation logic
- **Fix Applied:** Historical recomputation and logic correction

Key Learnings

- Drift is more dangerous than spikes
- Incremental pipelines need reconciliation
- Daily correctness ≠ historical correctness
- Baseline recomputation is a powerful diagnostic

Core Principle Reinforced

If metrics drift slowly, trust your baselines—not your assumptions.

Scenario 7

Late-Arriving Data Breaks Daily Aggregates

Problem Statement

A daily ETL pipeline aggregates data from multiple source tables, but **some sources update late**, causing **incomplete daily aggregates.** While the job can still run within the **2-hour SLA, business reporting requires full and accurate data,** and downstream consumers expect completeness.

Key Details

- Some source tables arrive late
- Daily aggregates depend on all sources
- Downstream consumers expect complete data
- Metrics used for business reporting
- SLA: 2 hours

Expected vs Actual Behavior

Expected	**Actual**
All source data included	Some sources missing
Accurate daily aggregates	Partial aggregates
Trusted business metrics	Misleading reports
SLA met with confidence	SLA met but data wrong

This is a **data completeness issue,** not a performance or execution failure.

Why This Problem Is Common

Because:

- Not all upstream systems follow the same schedule
- Late data is normal in distributed systems
- ETL pipelines often assume on-time arrivals

Running the job "on schedule" can still produce **incorrect results.**

Clarifying Questions

Before acting, a senior engineer asks:

- Which source tables arrive late, and how often?
- Are delays predictable or sporadic?
- Can aggregates be corrected later?
- Do consumers need final or provisional numbers?
- Is backfilling supported?

These questions focus on **accuracy guarantees,** not just timeliness.

Confirmed Facts & Assumptions

After investigation:

- Late-arriving tables are consistent offenders
- Aggregates miss a known subset of data
- Defaults or placeholders would distort metrics
- Re-running without new data doesn't help
- Accuracy is more important than speed

Interpretation:
This is a **pipeline design gap for late-arriving data**.

What the Pipeline Assumes vs Reality

Assumption	Reality
All data arrives on time	Some data arrives late
Running on schedule is enough	Accuracy requires completeness
Defaults are acceptable	Defaults mislead
SLA equals success	Correctness matters more

ETL pipelines must explicitly handle **data arrival variability.**

Root Cause Analysis

Step 1: Identify Late Sources

Observed:

- Certain tables consistently update after ETL start time
- Aggregates exclude those records

Conclusion:
The pipeline starts before all required data is available.

Step 2: Evaluate Aggregate Semantics

Observed:

- Aggregates are final, not provisional
- No correction or backfill mechanism exists

This confirms a **design mismatch between scheduling and data availability.**

Step 3: Conceptual Root Cause

The root cause is **lack of late-arriving data handling:**

- Pipeline assumes synchronized sources
- No delay or completeness check
- Aggregates produced prematurely

This is a **data orchestration issue,** not a compute issue.

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Run ETL anyway
- Fill missing data with defaults
- Ignore late arrivals

Right Approach

- Delay processing until all data arrives
- Add data availability checks
- Design for controlled delays

Senior engineers prioritize **correctness over punctuality.**

Step 5 : Validation of Root Cause

To confirm:

- Delay job start until late sources arrive
- Recompute aggregates
- Compare results with Expected totals

Outcome:
Aggregates become complete and reliable.

Step 6 :Corrective Actions

- Add data availability or watermark checks
- Delay job start when sources are incomplete
- Implement backfill support if needed
- Track and alert on late-arriving sources
- Communicate data freshness expectations

These steps ensure accurate daily reporting.

Step 7 : Result After Fix

Before Fix	**After Fix**
Incomplete aggregates	Complete aggregates
Misleading metrics	Accurate reporting
Reactive fixes	Predictable pipeline
SLA at risk	SLA and accuracy aligned

Final Resolution

- **Root Cause:** ETL ran before all source data arrived
- **Fix Applied:** Delayed processing until data completeness was ensured

Key Learnings

- Late data is normal, not exceptional
- Accuracy beats speed for business metrics
- ETL schedules must match data availability
- Completeness checks are essential

Core Principle Reinforced

A timely report with wrong data is worse than a slightly delayed report with correct data.

Scenario 8

Data Corruption During File Transfer

Problem Statement

An ETL pipeline detects **corrupted records during file transfer from S3 to the warehouse.** The data is **critical for downstream reporting**, the **1-hour SLA** is tight, and **re-fetching the entire dataset immediately is not feasible.**

Key Details

- Corrupted records detected during transfer
- Source: S3 → Data Warehouse
- Downstream reporting depends on this data
- Full re-fetch not immediately possible
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
Files transfer intact	Corrupted records detected
Data integrity preserved	Partial data corruption
Reports trustworthy	Metrics at risk
SLA met with confidence	SLA at risk due to data quality

This is a **data integrity failure,** not a scheduling or performance issue.

Why This Problem Is Dangerous

Because

- The job may still partially succeed
- Corruption may affect only some files
- Skipping bad records looks “safe” under SLA pressure

But **skipping corrupted data creates silent gaps that are hard to detect later.**

Clarifying Questions

Before acting, a senior engineer asks:

- Which files or partitions are corrupted?
- Is corruption repeatable or transient?
- Are checksums or validation available?
- Can only affected files be re-transferred?
- Will downstream tolerate partial data?

These questions focus on **data integrity preservation,** not speed alone.

Confirmed Facts & Assumptions

After investigation:

- Corruption is limited to specific files
- Transfer errors occurred during ingestion
- Skipping records would reduce totals
- Full dataset re-fetch is too slow
- Partial re-transfer is feasible

Interpretation:

This is a **transfer-level corruption issue**, not a source data issue.

What the Pipeline Assumes vs Reality

Assumption	Reality
File transfer is reliable	Transfers can corrupt
Job success implies clean data	Partial corruption possible
Skipping bad rows is acceptable	Skipping hides data loss
SLA matters most	Integrity matters more

ETL pipelines must validate **data correctness**, not just completion.

Root Cause Analysis

Step 1: Identify Corrupted Files

Observed:

- Checksum or parse failures on specific files
- Corruption isolated to a subset of data

Conclusion:
Corruption occurred during transfer, not at source.

Step 2: Assess Recovery Scope

Observed:

- Only affected files need re-transfer
- Full dataset re-fetch unnecessary

This enables **targeted recovery.**

Step 3: Conceptual Root Cause

The root cause is **lack of guaranteed integrity during transfer:**

- No end-to-end validation
- Partial file corruption
- Data integrity at risk

This is a **data movement reliability issue.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Skip corrupted records
- Ignore transfer errors
- Notify without recovery

Right Approach

- Re-transfer only affected files
- Validate integrity before load
- Preserve complete data

Senior engineers prioritize **correctness over convenience.**

Step 5 : Validation of Root Cause

To confirm:

- Re-transfer corrupted files
- Re-run integrity checks
- Compare record counts and hashes

Outcome:
All records load correctly without data loss.

Step 6 :Corrective Actions

- Re-transfer affected files only
- Enable checksum or hash validation
- Fail jobs on corruption detection
- Add alerts for transfer errors
- Log and audit corrupted file incidents

These steps prevent silent data corruption.

Step 7 : Result After Fix

Before Fix	**After Fix**
Corrupted data	Clean data
Risk of silent loss	Verified integrity
Broken trust	Restored trust
Reactive handling	Controlled recovery

Final Resolution

- **Root Cause:** Data corruption during file transfer
- **Fix Applied:** Targeted re-transfer with integrity validation

Key Learnings

- Data transfer is a failure point
- Corruption can be partial and silent
- Skipping bad data is dangerous
- Integrity checks are essential

Core Principle Reinforced

Never skip corrupted data—recover it, or you'll corrupt trust.

Scenario 9

ETL Job Succeeds but Business KPIs Are Wrong

Problem Statement

A daily ETL job **completes successfully,** but **key business KPIs—especially revenue—are off by nearly 20%**. Dashboards are live, multiple data sources feed the KPI, and the **2-hour SLA** leaves limited time to react.

Key Details

- ETL job reports success
- Revenue KPI deviates ~20%
- Multiple source systems involved
- Dashboards already consuming data
- SLA: 2 hours

Expected vs Actual Behavior

Expected	Actual
ETL success implies correct KPIs	ETL success but KPIs wrong
Revenue aligns with business Reality	Revenue significantly off
Dashboards trustworthy	Dashboards misleading
SLA met with confidence	SLA met but data incorrect

This is a **business logic failure,** not an execution failure.

Why This Problem Is Dangerous

Because:

- Technical monitoring shows green
- Pipelines didn't fail
- Errors surface only at the business layer

Teams often react by:

- Re-running the job
- Restarting pipelines
- Informing stakeholders immediately

But **re-running correct code does not fix incorrect logic.**

Clarifying Questions

Before acting, a senior engineer asks:

- Which transformation defines the KPI?
- Did any calculation logic change recently?
- Are joins, filters, or aggregations applied correctly?
- Are all contributing sources aligned in grain and timing?
- Do intermediate metrics match expectations?

These questions focus on **semantic correctness,** not pipeline health.

Confirmed Facts & Assumptions

After investigation:

- Source data volumes look correct
- KPI divergence appears after transformation layer
- Aggregation or filter logic is inconsistent
- Re-running produces the same wrong numbers
- Job success masks logical errors

Interpretation:
This is a **faulty KPI calculation**, not missing or late data.

What the Pipeline Assumes vs Reality

Assumption	**Reality**
Job success = correct output	Logic can be wrong
Source data errors cause KPI issues	Transformations introduce errors
Re-run fixes inconsistencies	Re-run repeats the mistake
Monitoring catches issues	Business logic unvalidated

ETL pipelines must validate **meaning**, not just movement.

Root Cause Analysis

Step 1: Trace KPI Calculation Path

Observed:

- KPI derived from multiple joins and aggregations
- Intermediate values diverge from Expected totals

Conclusion:
Error introduced during transformation logic.

Step 2: Audit Business Logic

Observed:

- Incorrect filter condition
- Misaligned join grain
- Aggregation double-counts some records

This confirms **business logic misimplementation.**

Step 3: Conceptual Root Cause

The root cause is **lack of business-rule validation:**

- KPI logic not audited regularly
- No reconciliation against known benchmarks
- ETL correctness assumed equals business correctness

This is a **semantic validation gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Re-run ETL job
- Ignore the deviation
- Notify stakeholders without diagnosis

Right Approach

- Audit KPI business logic
- Validate transformations step-by-step
- Reconcile against trusted benchmarks

Senior engineers debug **meaning before mechanics.**

Step 5 : Validation of Root Cause

To confirm:

- Fix transformation logic
- Recompute KPI
- Compare with Expected business numbers

Outcome:
KPIs align with real-world business performance.

Step 6 :Corrective Actions

- Audit and document KPI logic
- Add reconciliation checks for key metrics
- Validate intermediate aggregates
- Alert on KPI deviations, not just job failures
- Review KPI logic after every change

These steps prevent silent business impact.

Step 7 : Result After Fix

Before Fix	**After Fix**
KPIs off by 20%	KPIs accurate
Misleading dashboards	Trusted dashboards
Re-run confusion	Clear diagnosis
Business risk	Business confidence

Final Resolution

- **Root Cause:** Incorrect business logic in KPI transformations
- **Fix Applied:** Audited and corrected KPI calculation logic

Key Learnings

- ETL success ≠ business correctness
- KPIs must be validated, not assumed
- Logic bugs are more dangerous than job failures
- Always reconcile metrics with Reality

Core Principle Reinforced

A green pipeline with wrong KPIs is still a production failure.

Scenario 10

Data Format Changes Break Downstream ETL Jobs

Problem Statement

An upstream system changes its data format from **CSV to JSON**, causing downstream ETL pipelines to **fail during parsing.** Multiple dependent pipelines are blocked, rollback is impossible, and the **1-hour SLA** is under immediate risk.

Key Details

- Upstream data format change: CSV → JSON
- ETL parsing failures
- Multiple downstream dependencies
- Rollback not possible
- SLA: 1 hour

Expected vs Actual Behavior

Expected	**Actual**
ETL parses incoming data correctly	Parsing fails
Downstream pipelines continue	Pipelines blocked
Data format changes detected	Change discovered at runtime
SLA protected	SLA breached

This is a **data contract and format compatibility failure**, not a compute issue.

Why This Problem Is Common

Because:

- Upstream teams evolve formats for flexibility
- Format changes are often deployed independently
- ETL pipelines assume fixed file formats

Without validation, format changes surface **only after failure.**

Clarifying Questions

Before acting, a senior engineer asks:

- Is the new JSON format backward compatible?
- Are all files migrated or mixed-format?
- Can format detection be automated?
- Do downstream pipelines expect strict schema?
- Is dual-format support required temporarily?

These questions focus on **compatibility strategy,** not just recovery.

Confirmed Facts & Assumptions

After investigation:

- All new files are JSON
- ETL parser expects CSV
- Downstream pipelines are blocked
- Waiting for upstream rollback is not an option
- Updating parser unblocks all consumers

Interpretation:
This is a **format evolution handling gap**.

What the Pipeline Assumes vs Reality

Assumption	**Reality**
Input format is static	Format evolves
Parsing logic rarely changes	Parsing must adapt
Upstream will coordinate changes	Changes may be unilateral
Failures are rare	Format drift is common

ETL pipelines must be **format-aware and resilient.**

Root Cause Analysis

Step 1: Identify Point of Failure

Observed:

- Parser fails at read stage
- No transformation logic executed

Conclusion:
Failure caused purely by incompatible input format.

Step 2: Evaluate Recovery Options

Options reviewed:

- Skipping files (data loss)
- Waiting on upstream (SLA breach)
- Updating parser (correct)

This confirms the **only viable recovery path.**

Step 3: Conceptual Root Cause

The root cause is **rigid input parsing logic:**

- ETL assumes CSV-only inputs
- No format detection or flexibility
- Format change breaks entire pipeline

This is a **pipeline robustness issue.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Skip changed files
- Ignore parsing errors
- Wait only on upstream teams

Right Approach

- Update ETL parser to handle new format
- Add format detection and validation
- Monitor for format changes

Senior engineers design for **format evolution,** not stability assumptions.

Step 5 : Validation of Root Cause

To confirm:

- Update parser to support JSON
- Re-run ETL
- Verify downstream pipelines recover

Outcome:
All pipelines resume successfully.

Step 6 :Corrective Actions

- Update ETL parsing logic
- Support dual formats temporarily if needed
- Add format validation at ingestion
- Alert on unExpected format changes
- Document data contracts clearly

These steps prevent similar failures in the future.

Step 7 : Result After Fix

Before Fix	**After Fix**
Parsing failures	Successful ingestion
Downstream blocked	Pipelines unblocked
SLA breached	SLA met
Reactive fixes	Resilient design

Final Resolution

- **Root Cause:** ETL parser incompatible with new input format
- **Fix Applied:** Updated parsing logic to handle JSON

Key Learnings

- Format changes are inevitable
- Parsing is a critical failure point
- Skipping data hides real problems
- Resilient ingestion prevents outages

Core Principle Reinforced

Design ETL pipelines to adapt to format changes—not break because of them.

Scenario 11

Duplicate Loads Introduced After ETL Retry

Problem Statement

An ETL job experiences a **transient failure** and is retried. While the retry completes successfully, the warehouse now contains **duplicate rows**, causing **business metrics to be inflated.** There is **no automated deduplication,** and the **2-hour SLA** is under pressure.

Key Details

- ETL job retried after transient failure
- Duplicate rows introduced in warehouse
- Business metrics highly sensitive to duplication
- No idempotency or deduplication in place
- SLA: 2 hours

Expected vs Actual Behavior

Expected	Actual
Retry safely resumes processing	Retry re-inserts same data
Data loaded exactly once	Duplicate rows created
Business metrics remain accurate	Metrics inflated
SLA met with confidence	SLA met but data incorrect

This is a **pipeline design flaw,** not a runtime failure.

Why This Problem Is Dangerous

Because:

- Retries are Expected and normal
- Job finishes “successfully”
- Duplication often goes unnoticed initially

Teams often respond by:

- Manually deleting duplicates
- Re-running jobs
- Ignoring small discrepancies

But **manual cleanup does not prevent the next failure from recreating the same issue.**

Clarifying Questions

Before acting, a senior engineer asks:

- What defines a unique record in this dataset?
- Are writes append-only or overwrite-based?
- Are primary keys enforced at load time?
- Does the pipeline track processed records?
- Can the job be safely re-run multiple times?

These questions focus on **retry safety**, not quick cleanup.

Confirmed Facts & Assumptions

After investigation:

- Retry reprocessed the same input data
- Warehouse allows duplicate inserts
- No natural or enforced primary key
- Manual deletion fixes data only temporarily
- Next retry will cause duplication again

Interpretation:
This is a **non-idempotent ETL design**.

What the Pipeline Assumes vs Reality

Assumption	Reality
Retries are harmless	Retries duplicate data
Job success ensures correctness	Logic allows duplication
Manual fixes are acceptable	Manual fixes don't scale
Failures are rare	Retries are inevitable

ETL pipelines must be **retry-safe by design.**

Root Cause Analysis

Step 1: Analyze Load Semantics

Observed:

- Inserts are unconditional
- No merge, upsert, or overwrite logic
- Same records loaded again on retry

Conclusion:
The pipeline cannot distinguish new data from already-processed data.

Step 2: Inspect Retry Behavior

Observed:

- Retry starts from same input
- No checkpointing or watermark enforcement
- No uniqueness constraint in target

This confirms **non-idempotent writes.**

Step 3: Conceptual Root Cause

The root cause is **lack of idempotency:**

- ETL does not guarantee “exactly once” behavior
- Retries reinsert the same records
- Data correctness depends on manual intervention

This is a **reliability and design issue**, not an operational mistake.

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Manually delete duplicates
- Re-run jobs again
- Ignore minor duplication

Right Approach

- Implement idempotent ETL logic
- Use upserts / merges / overwrite-by-key
- Enforce uniqueness at load time

Senior engineers design pipelines that **can be re-run safely at any time.**

Step 5 : Validation of Root Cause

To confirm:

- Implement idempotent logic
- Retry the job intentionally
- Verify no new duplicates appear

Outcome:
Retries no longer affect data correctness.

Step 6 :Corrective Actions

- Define natural or synthetic primary keys
- Use merge/upsert instead of blind inserts
- Track processed records or checkpoints
- Add duplicate-detection checks
- Test retry scenarios explicitly

These steps make pipelines resilient to failures.

Step 7 : Result After Fix

Before Fix	After Fix
Duplicate rows	Exactly-once loads
Manual cleanup	Automated correctness
Retry risk	Retry-safe pipeline
Business trust eroded	Business trust restored

Final Resolution

- **Root Cause:** Non-idempotent ETL design
- **Fix Applied:** Implemented idempotent load logic

Key Learnings

- Retries are not edge cases—they are normal
- Idempotency is a core ETL requirement
- Manual fixes don't scale
- Exactly-once behavior must be designed

Core Principle Reinforced

If an ETL job can't be re-run safely, it's not production-ready.

Scenario 12

Missing Partitions Due to Upstream Failure

Problem Statement

A daily ETL pipeline expects **one partition per day**, but an **upstream job failed the previous day**, resulting in a **missing partition.** Downstream reporting depends on complete data, the **2-hour SLA** is tight, and while recovery is possible, it is time-consuming.

Key Details

- Daily partitioned data model
- Upstream job failure caused missing partition
- Downstream reporting requires full data
- Recovery possible but time-consuming
- SLA: 2 hours

Expected vs Actual Behavior

Expected	**Actual**
All daily partitions present	One partition missing
Aggregates computed correctly	Aggregates incomplete
Reports trusted	Reports misleading
SLA met with confidence	SLA at risk

This is a **data dependency failure,** not a transformation error.

Why This Problem Is Risky

Because:

- ETL job itself did not fail
- Missing data may not raise immediate errors
- Dashboards still render successfully

Teams often:

- Skip the missing partition to meet SLA
- Fill gaps with defaults
- Ignore the issue temporarily

But **silent gaps permanently damage data trust.**

Clarifying Questions

Before acting, a senior engineer asks:

- Which partition is missing and why?
- Can upstream data be reprocessed safely?
- How critical is this partition for reporting?
- Is backfill supported without side effects?
- Can recovery complete within SLA?

These questions prioritize **completeness over convenience.**

Confirmed Facts & Assumptions

After investigation:

- One specific date partition is missing
- Upstream failure is the root cause
- Skipping partition reduces totals
- Defaults would distort metrics
- Upstream recovery can restore data

Interpretation:
This is a **dependency recovery issue**, not an ETL bug.

What the Pipeline Assumes vs Reality

Assumption	Reality
Upstream always succeeds	Upstream failures happen
All partitions exist	Partitions can be missing
Skipping is acceptable	Skipping breaks accuracy
SLA equals success	Completeness matters more

ETL pipelines must treat **missing partitions as failures**, not warnings.

Root Cause Analysis

Step 1: Identify Missing Partition

Observed:

- Partition for a specific date absent
- ETL logic expects it unconditionally

Conclusion:
Upstream data did not arrive.

Step 2: Assess Impact of Skipping

Observed:

- Aggregates lower than Expected
- No clear error signals

Skipping creates **silent data loss.**

Step 3: Conceptual Root Cause

The root cause is **unhandled upstream dependency failure:**

- ETL does not verify partition completeness
- Missing data allowed through
- Downstream metrics silently degrade

This is a **pipeline dependency management gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Skip missing partition
- Fill with defaults
- Ignore the gap

Right Approach

- Trigger upstream recovery
- Backfill missing partition
- Validate completeness before proceeding

Senior engineers **block pipelines on missing dependencies.**

Step 5 : Validation of Root Cause

To confirm:

- Trigger upstream recovery
- Re-run ETL for missing partition
- Validate record counts

Outcome:
Data completeness restored and metrics correct.

Step 6 :Corrective Actions

- Add partition completeness checks
- Block ETL when required partitions are missing
- Automate upstream recovery or backfill
- Alert on missing partitions
- Document dependency contracts

These steps prevent silent data gaps.

Step 7 : Result After Fix

Before Fix	**After Fix**
Missing partition	Complete dataset
Incomplete metrics	Accurate metrics
Silent failure	Explicit recovery
Trust erosion	Trust restored

Final Resolution

- **Root Cause:** Upstream failure caused missing partition
- **Fix Applied:** Triggered upstream recovery and backfill

Key Learnings

- Upstream dependencies must be verified
- Missing partitions are data failures
- Defaults hide real problems
- Recovery beats skipping

Core Principle Reinforced

If upstream data is missing, stop and recover—never silently move forward.

Scenario 13

Aggregation Skew Produces Incorrect Business Metrics

Problem Statement

An ETL job performs aggregations to compute business KPIs, but **one dominant key (hot key)** accounts for a disproportionate share of the data. As a result, **aggregated metrics become misleading**, downstream dashboards show distorted values, and the **1-hour SLA** is under pressure.

Key Details

- One key dominates aggregation workload
- High data volume
- Aggregations used directly in dashboards
- ETL job completes but metrics are misleading
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
Aggregations evenly distributed	One key dominates results
Metrics reflect true business state	Metrics skewed by hot key
Dashboards trustworthy	Dashboards misleading
SLA met with confidence	SLA met but data wrong

This is a **data correctness issue caused by skew,** not a job failure.

Why This Problem Is Subtle

Because:

- ETL job does not fail
- Aggregation logic is syntactically correct
- Performance may degrade only slightly

Teams often:

- Re-run the job
- Scale the cluster
- Assume data distribution is "normal"

But **skew silently breaks both performance and correctness.**

Clarifying Questions

Before acting, a senior engineer asks

- Which key dominates the aggregation?
- Is this dominance Expected or anomalous?
- Are KPIs supposed to weight this key equally?
- Does aggregation logic assume uniform distribution?
- Can the skewed key be handled separately?

These questions focus on **data distribution and semantics**, not infrastructure.

Confirmed Facts & Assumptions

After investigation:

- One key contributes an outsized portion of records
- Aggregation logic treats all keys equally
- Metrics heavily influenced by that key
- Re-running produces identical skew
- Scaling compute does not change distribution

Interpretation:
This is an **aggregation design issue**, not a compute limitation.

What the Pipeline Assumes vs Reality

Assumption	**Reality**
Keys are evenly distributed	One hot key dominates
Aggregation logic is neutral	Skew distorts results
Scaling fixes issues	Scaling doesn't fix skew
Job success implies correctness	Metrics still wrong

ETL pipelines must account for **data skew explicitly.**

Root Cause Analysis

Step 1: Identify Skewed Key

Observed:

- One key contributes a majority of records
- Aggregated totals driven primarily by this key

Conclusion:
The aggregation is dominated by a hot key.

Step 2: Evaluate Aggregation Semantics

Observed:

- No special handling for dominant keys
- No pre-aggregation or normalization

This confirms **skew-aware logic is missing.**

Step 3: Conceptual Root Cause

The root cause is **unhandled aggregation skew:**

- Hot key overwhelms aggregation
- Metrics lose representativeness
- Dashboards mislead stakeholders

This is a **data modeling and aggregation strategy gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Re-run the job
- Scale the cluster
- Ignore skew

Right Approach

- Pre-aggregate the skewed key
- Normalize or separate hot-key handling
- Balance aggregation logic

Senior engineers design aggregations based on **data distribution,** not assumptions.

Step 5 : Validation of Root Cause

To confirm:

- Pre-aggregate or isolate the skewed key
- Recompute metrics
- Compare with Expected business value

Outcome:
Metrics align with real business behavior.

Step 6 :Corrective Actions

- Identify and isolate hot keys
- Pre-aggregate skewed dimensions
- Validate KPI sensitivity to dominant keys
- Add distribution checks before aggregation
- Monitor skew trends over time

These steps prevent distorted metrics at scale.

Step 7 : Result After Fix

Before Fix	**After Fix**
Skewed metrics	Balanced metrics
Dashboards misleading	Dashboards trustworthy
Reactive debugging	Skew-aware design
SLA risk	SLA protected

Final Resolution

- **Root Cause:** Aggregation skew caused by a dominant key
- **Fix Applied:** Pre-aggregation and skew-aware aggregation logic

Key Learnings

- Data skew affects correctness, not just performance
- Hot keys must be handled explicitly
- Scaling compute doesn't fix skew
- Aggregation logic must reflect data Reality

Core Principle Reinforced

If one key dominates your data, it will dominate your metrics—unless you design for it.

Scenario 14

ETL Job Loads Data from the Wrong Source Table

Problem Statement

An ETL job completes successfully, but due to a **misconfigured job configuration,** it reads from a **stale or incorrect source table.** As a result, **incorrect data is loaded into the warehouse,** impacting multiple downstream reports. A **quick and accurate fix** is required within the **1-hour SLA.**

Key Details

- Job configuration points to wrong source table
- ETL job completes without failure
- Multiple downstream reports impacted
- Incorrect data already visible to users
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
ETL reads latest source table	ETL reads stale table
Warehouse reflects correct data	Warehouse contains outdated data
Reports trustworthy	Reports misleading
SLA met with confidence	SLA met but trust broken

This is a **configuration correctness failure,** not a processing or logic error.

Why This Problem Is Risky

Because:

- Pipeline runs “successfully”
- No technical errors are raised
- Issue is detected only through business validation

Teams sometimes:

- Manually patch data
- Notify stakeholders without fixing root cause
- Ignore until next run

But **misconfiguration silently undermines data trust faster than hard failures.**

Clarifying Questions

Before acting, a senior engineer asks:

- Which source table was Expected vs configured?
- When was the configuration last changed?
- Are environment-specific configs validated?
- How much downstream data is impacted?
- Can a clean re-run overwrite bad data safely?

These questions focus on **restoring correctness quickly and safely.**

Confirmed Facts & Assumptions

After investigation:

- Job config references an outdated table
- No transformation logic error exists
- Data is consistently wrong across reports
- Manual patching would be error-prone
- Re-running with correct config fixes data

Interpretation:
This is a **configuration drift or validation gap**.

What the Pipeline Assumes vs Reality

Assumption	**Reality**
Configurations are always correct	Configs can drift
Job success implies correct source	Wrong source still succeeds
Manual fixes are acceptable	Manual fixes don't scale
Monitoring catches errors	Config issues go unnoticed

ETL pipelines must validate **inputs, not just execution.**

Root Cause Analysis

Step 1: Verify Source Configuration

Observed:

- Source table name mismatched Expected value
- No validation step caught the mismatch

Conclusion:
ETL read from an unintended source.

Step 2: Assess Impact Scope

Observed:

- All downstream reports affected
- Data consistently incorrect, not partial

This confirms a **systemic configuration issue.**

Step 3: Conceptual Root Cause

The root cause is **lack of configuration validation:**

- Wrong source table referenced
- No guardrails or sanity checks
- Incorrect data propagated silently

This is a **trust and governance failure,** not a compute issue.

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore incorrect data
- Manually patch warehouse tables
- Only notify stakeholders

Right Approach

- Fix configuration
- Re-run ETL with correct source
- Validate results before release

Senior engineers restore **trust through correctness**, not communication alone.

Step 5 : Validation of Root Cause

To confirm:

- Update configuration to correct table
- Re-run ETL
- Validate row counts and key metrics

Outcome:

Warehouse data aligns with **Expected** business **Reality**.

Step 6 :Corrective Actions

- Re-run ETL with correct source table
- Add configuration validation checks
- Use environment-specific config controls
- Log source metadata per run
- Alert on unExpected source changes

These steps prevent repeat incidents.

Step 7 : Result After Fix

Before Fix	After Fix
Incorrect data	Correct data
Broken trust	Trust restored
Manual intervention risk	Automated correction
SLA at risk	SLA met

Final Resolution

- **Root Cause:** ETL misconfigured to read from stale source table
- **Fix Applied:** Corrected configuration and re-ran ETL

Key Learnings

- ETL failures aren't always technical
- Configuration errors are high-impact
- Job success ≠ correct inputs
- Validation should include source checks

Core Principle Reinforced

A perfectly running pipeline can still produce perfectly wrong data—validate your sources.

Scenario 15

ETL Job Passes Validation but Data Semantics Are Broken

Problem Statement

An ETL pipeline completes successfully and **all automated validation checks pass** (schema, nulls, ranges, counts). However, **business users report that the data meaning is wrong**, caused by an **upstream miscalculation.** Dashboards are live, trust is at risk, and the **2-hour SLA** is ticking.

Key Details

- All technical validations pass
- Data values are syntactically correct
- Business meaning is incorrect
- Upstream miscalculation involved
- SLA: 2 hours

Expected vs Actual Behavior

Expected	**Actual**
Validation guarantees correctness	Validation passes, meaning wrong
Dashboards reflect business Reality	Dashboards misleading
Data trusted by stakeholders	Trust eroded
SLA met with confidence	SLA met but impact high

This is a **semantic data failure**, not a pipeline failure.

Why This Problem Is the Most Dangerous

Because:

- Every automated check is green
- Pipelines look "healthy"
- Errors surface only through business intuition

Teams often:

- Re-run the job
- Assume edge cases
- Delay action because validation passed

But **passing validation does not guarantee correct meaning**.

Clarifying Questions

Before acting, a senior engineer asks:

- What business rule defines correctness?
- Which upstream calculation changed?
- Do derived metrics align with business expectations?
- Are semantic assumptions documented?
- What checks would have caught this earlier?

These questions focus on **meaning, not mechanics.**

Confirmed Facts & Assumptions

After investigation:

- Schema and data quality checks all pass
- Values conform to Expected ranges
- Upstream logic changed subtly
- Derived metrics violate business interpretation
- Re-running reproduces the issue

Interpretation:
This is a **missing semantic validation problem**.

What the Pipeline Assumes vs Reality

Assumption	**Reality**
Technical checks are sufficient	Semantics can still break
Correct shape = correct meaning	Shape ≠ meaning
Validation equals trust	Trust requires context
Automation catches everything	Humans still matter

ETL pipelines often validate **form**, not **intent**.

Root Cause Analysis

Step 1: Identify Semantic Mismatch

Observed:

- Numbers are valid but interpreted incorrectly
- KPIs contradict known business behavior

Conclusion:
Business rules are not encoded in validation.

Step 2: Trace Upstream Calculations

Observed:

- Calculation logic changed without semantic checks
- No contract enforcing business meaning

This confirms a **semantic drift.**

Step 3: Conceptual Root Cause

The root cause is **absence of semantic validation:**

- Pipeline validates structure, not intent
- Upstream changes alter meaning silently
- Data correctness is assumed, not enforced

This is a **data trust and governance gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Re-run ETL
- Ignore because validation passed
- Notify stakeholders without fixing

Right Approach

- Add semantic validation rules
- Encode business meaning into checks
- Validate outputs against expectations

Senior engineers protect **trust**, not just pipelines.

Step 5 : Validation of Root Cause

To confirm:

- Implement semantic checks (e.g., ratios, trend bounds)
- Re-run validation
- Observe failure where semantics break

Outcome:

Semantic errors are caught before dashboards update.

Step 6 :Corrective Actions

- Add business-rule-based validations
- Validate KPI relationships and ratios
- Monitor trend-level anomalies
- Involve domain experts in validation design
- Version and document semantic rules

These steps prevent “green but wrong” pipelines.

Step 7 : Result After Fix

Before Fix	**After Fix**
Technically valid, semantically wrong	Technically and semantically correct
Dashboards misleading	Dashboards trusted
Hidden trust erosion	Early detection
Reactive investigation	Proactive governance

Final Resolution

- **Root Cause:** Missing semantic validation for business meaning
- **Fix Applied:** Added semantic and business-rule validations

Key Learnings

- Validation ≠ correctness
- Semantics matter more than structure
- Business rules must be codified
- Data trust is engineered, not assumed

Core Principle Reinforced

If your pipeline validates shape but not meaning, it will eventually betray trust.