DATA PIPELINE FAILURES

Real Interview Scenarios & How to Handle Them

A practical guide for Data Engineers to answer real-world data pipeline failure questions with confidence

By Ankita Gulati

Senior Data Engineer | Interview Mentor

Interview Edition • Practical • Real Scenarios

Table Of Content

Scenario 1
Daily Spark Runtime Explodes

Problem Statement

You are responsible for a daily Spark batch job that processes approximately 1 TB of data.

- Normal runtime: 20 minutes
- Current runtime: 90 minutes
- Downstream business reports are blocked

The job does not fail but runs significantly slower than usual

Objective: Diagnose the root cause and bring the job back within the 30-minute SLA.

Clarifying Questions

Before proposing solutions, the engineer must reduce ambiguity.
Key questions include:

- Did the data volume or distribution change?
- Were there schema, join, or aggregation changes?
- Are all Spark stages slow or only specific ones?
- Do some tasks take much longer than others?
- Are executors underutilized or stuck waiting?

Clarifying Information & Assumptions

After initial investigation, the following facts are confirmed:

- Data volume increased by ~15%
- No changes in code, schema, or Spark version
- Cluster configuration remains unchanged
- No job failures or retries occurred

These assumptions indicate the slowdown is **not expected growth-related behavior**.

Key Observation

A **15% increase in data volume** should result in a small, near-linear increase in runtime.
A 4–5× runtime increase strongly suggests:

- Uneven data distribution
- Execution imbalance across Spark tasks
- Presence of straggler tasks

This shifts focus from scaling resources to execution behavior analysis.

Solution: Step-by-Step Deep Explanation

Step 1: Analyze Job Execution Using Spark UI

The first step is to inspect how Spark is executing the job, not how much data it

processes. Key indicators:

- Large gap between average task time and maximum task time
- One or two tasks running significantly longer than others
- Several executors idle while a few remain busy

This pattern confirms that a small subset of tasks is dominating total runtime.

Step 2: Understand Why Straggler Tasks Occur

Spark divides data into **partitions**, and each task processes one partition.

Stragglers occur when:

- Some partitions contain **disproportionately large data**
- Those partitions take much longer to process
- The entire waits for the slowest task to finish

This behavior is a classic symptom of **data skew**.

Step 3: What Data Skew Means Conceptually

Data skew happens when:

- A small number of keys appear extremely frequently
- Joins or aggregations group large volumes of data under those keys
- Spark assigns those heavy keys to one or few

partitions As a result:

- Parallelism collapses
- Most executors finish early
- One executor becomes the bottleneck for the entire job.

Step 4: Why Scaling Infrastructure Is Not the Correct First Fix

Increasing cluster size or executor memory does **not** redistribute skewed data.

Consequences of scaling:

- Skewed partitions still execute on a single executor
- Additional executors remain underutilized
- Cloud cost increases without fixing the root cause

Scaling hides the problem instead of solving it

Step 5: Diagnosing Skewed Keys

To confirm skew:

- Inspect join keys and group-by columns
- Analyze frequency distribution of keys
- Identify extreme outliers with very high record counts

This step identifies **where the imbalance originates**.

Step 6: Applying the Correct Fix

Once skew is confirmed, the objective is to redistribute heavy keys so work can be parallelized.

Key techniques:

- **Key Salting**: Splits heavy keys into multiple sub-keys, allowing parallel processing.
- **Repartitioning**: Ensures more even data distribution before expensive joins or aggregations.
- **Broadcast Joins**: Avoids shuffle when one side of the join is small, eliminating skew impact.

These techniques directly address partition imbalance, not hardware limitations.

Step 7: Result After Fix

After removing skew:

- Tasks complete at similar times
- Executors are evenly utilized
- Stage completion time drops significantly
- Overall job runtime returns within SLA

The solution is **durable and cost-efficient.**

Final Resolution

Root Cause:
Data skew causing straggler tasks.

Correct Resolution Path:

- Observe execution behavior
- Identify skewed partitions
- Apply targeted data redistribution
- Scale infrastructure only if required afterward

Key Learning Outcomes

From this scenario, a learner understands:

- Why performance issues are often data-driven
- How Spark execution behavior affects runtime
- Why scaling is not a first-response solution
- How to think like a senior data engineer under SLA pressure

Core Principle Reinforced

In distributed data systems, **fixing data distribution problems delivers better results than adding compute.**

Scenario 2

Intermittent Spark Job Failures

Problem Statement

You own a production Spark batch job that fails intermittently on specific days despite no changes to code, schema, or infrastructure. The job is business-critical, and failures block downstream analytics.

Key Details

- Same code and schema across runs
- Failures occur only on certain days
- Job is critical for downstream reports
- Restarts sometimes succeed
- No recent deployments or configuration changes

Clarifying Questions

Before proposing solutions, reduce ambiguity.
Key questions include:

- Do failures align with specific dates or data sources?
- Is the failure deterministic for a given day's input?
- Does the job fail at the same stage every time?
- Are error messages consistent across failed runs?
- Does rerunning with identical input always fail or sometimes pass?

Clarifying Information & Assumptions

After investigation, the following facts are confirmed:

- No code or schema changes were deployed
- Spark version and cluster configuration are unchanged
- Failures appear only on certain days
- Reruns sometimes succeed without any changes

Interpretation

This rules out deployment and infrastructure issues. Non-deterministic behavior with identical code strongly suggests input-dependent problems.

Key Observation

With stable code and infrastructure:

- Some daily runs fail
- Some succeed without intervention
- Retries occasionally work

This behavior is inconsistent with code bugs and strongly points toward **data variability across days**.

Investigation & Root Cause Analysis

Step 1: Observe Failure Patterns

- Failures occur at the same transformation stage
- Error messages vary depending on the input day
- The same job succeeds with adjacent days' data

Conclusion:

Execution logic is stable; input characteristics differ.

Step 2: Understand System Behavior

Spark assumes:

- Well-formed input
- Valid data types
- Predictable constraints

When these assumptions are violated:

- Parsing errors occur
- Null or type violations break transformations
- Rare edge cases crash the job

These issues surface only when problematic records exist.

Step 3: Conceptual Root Cause Explanation

Intermittent failures typically occur when:

- Upstream systems emit malformed or partial data
- Edge-case records appear infrequently
- Validation is weak or missing

Because Spark processes entire partitions:

- A small number of bad records can fail the whole job

Step 4 :Why Obvious Fixes Are Wrong

- **Adding retries**:
  Hides the problem and creates operational noise. Bad data will fail again.
- **Increasing timeouts**:
  Data quality issues are not time-related.
- **Ignoring failures**:
  Leads to SLA breaches, escalations, and loss of business trust.

These actions treat symptoms, not the root cause.

Step 5 :Validation of Root Cause

To confirm:

- Compare successful vs failed day inputs
- Inspect raw records from failed partitions
- Identify malformed rows, missing fields, or invalid values

Outcome:
Failed days contain inconsistent or corrupted input data.

Step 6:Corrective Actions

- Add data validation at ingestion
- Quarantine or drop bad records safely
- Enforce schema and constraint checks
- Add alerts for upstream data quality issues

This improves resilience without masking problems.

Step 7:Result After Fix

- No more intermittent failures
- Bad data is isolated instead of destructive
- Downstream analytics stabilize
- Operational confidence is restore

Final Resolution

- **Root Cause**: Inconsistent input data on specific days
- **Fix Applied**: Defensive ingestion with data validation

Key Learnings

- Intermittent failures are rarely random
- Stable code + unstable runs usually mean data issues
- Retries are not a root-cause solution
- Senior engineers debug inputs before tuning systems

Core Principle Reinforced

Reliable data systems are built by validating data, not by retrying failures.

Scenario 3
Upstream Dependency Failure Blocking the DAG

Problem Statement

A production ETL pipeline with multiple dependencies is stalled because an upstream job has failed. Downstream analytics are blocked, the SLA is one hour, and no immediate code changes are allowed on the upstream system.

Key Details

- Multi-stage DAG with upstream dependencies
- One upstream job failure blocks the pipeline
- Downstream analytics cannot proceed
- SLA breach risk within 1 hour

Upstream code cannot be changed immediately

Clarifying Questions

Before taking action, reduce ambiguity:
What exactly failed in the upstream job?

- Is the failure persistent or a one-time anomaly?
- Did the upstream schema, data volume, or format change?
- Are downstream jobs dependent on full or partial upstream data?
- Is there historical precedent for similar failures?

Clarifying Information & Assumptions

After investigation, the following facts are confirmed:

- Upstream job failed consistently for the same reason
- Failure occurs before data publication
- No partial or corrupted data was emitted
- Downstream jobs depend fully on upstream outputs
- SLA countdown is actively progressing

Interpretation:

This is a dependency integrity issue, not an orchestration glitch.

Key Observation

The DAG is stalled not due to scheduler failure, but because:

- Dependency guarantees are enforced correctly
- Running downstream jobs without upstream data would violate data correctness
- This confirms the issue is upstream correctness, not workflow orchestration.

Investigation & Root Cause Analysis

Step 1: Inspect Upstream Failure Logs

- Review error logs and stack traces
- Identify the exact failure point
- Check for schema mismatches or missing fields

Conclusion:
The upstream job is failing deterministically.

Step 2: Understand DAG Dependency Behavior

Workflow orchestrators (Airflow, Dagster, ADF, etc.):

- Enforce execution order to ensure data correctness
- Prevent downstream jobs from running on incomplete data

By design:

- A single upstream failure halts the DAG

Step 3: Conceptual Root Cause Explanation

Root cause patterns in dependency failures often include:

- Upstream schema changes
- Contract violations between producers and consumers
- Missing backward compatibility
- Even small upstream changes can cascade across the DAG.

Step 4:Why Obvious Fixes Are Wrong

- **Force downstream execution:**
  Produces incomplete or misleading metrics, causing silent data corruption.
- **Skipping failed tasks**:
  Temporarily unblocks execution but violates data contracts.
- **Notifying business only:**
  Maintains transparency but does not solve the technical issue.

These actions prioritize speed over correctness

Step 5: Validation of Root Cause

To confirm:

- Compare upstream schema before and after failure
- Validate downstream expectations against current upstream output
- Reproduce failure in isolation if possible

Outcome:
An upstream schema change caused the job failure.

Step 6: Corrective Actions

- Communicate findings to upstream owners immediately
- Apply schema reconciliation or compatibility handling downstream (if allowed)
- Add schema validation and contract checks between jobs
- Improve monitoring for upstream contract changes
- This preserves data integrity while preventing future incidents.

Step 7: Result After Fix

- Upstream issue is addressed
- DAG resumes normal execution
- Downstream analytics are accurate
- SLA compliance is restored

Final Resolution

- **Root Cause**: Upstream schema change causing dependency failure
- **Fix Applied**: Schema contract validation and coordinated upstream resolution

Key Learnings

- DAG dependencies exist to protect data correctness
- Bypassing dependencies causes silent data corruption
- Root-cause analysis starts upstream, not downstream
- Senior engineers protect data integrity under SLA pressure

Core Principle Reinforced

Correct data late is better than incorrect data on time.

Scenario 4

Partial Job Success Causing Silent Data Gaps

Problem Statement

A daily production data load completes successfully, but a subset of partitions fails to load, resulting in missing records in the data warehouse. The SLA is two hours, and partial data risks financial misreporting.

Key Details

- Job reports success at a high level
- Some partitions failed or were skipped
- Missing records in the warehouse
- Financial reporting depends on data accuracy
- Full rerun is expensive and time-consuming

Clarifying Questions

Before taking action, a senior engineer asks:

- Which partitions failed and why?
- Was the failure detected or silent?
- Are downstream consumers reading partial data?
- Can failed partitions be isolated and replayed safely?
- Is idempotency guaranteed for reprocessing?

Confirmed Facts & Assumptions

After investigation:

- Only a small subset of partitions failed
- Most data loaded correctly
- Failures were due to transient issues
- Full job rerun would duplicate already processed data
- Downstream systems do not expect partial datasets

Interpretation:

This is not a full pipeline failure, but a **partial execution integrity issue**.

Key Observation

The most dangerous aspect is not the failure itself, but that:

- The job completed
- Partial data looks “successful”
- Missing records can silently corrupt reports

This makes partial success more dangerous than a full failure.

Root Cause Analysis

Step 1: Identify Failed Partitions

- Inspect job logs and execution metadata
- Locate partitions with write or commit failures
- Verify warehouse table partition completeness

Conclusion:
The failure is localized, not systemic.

Step 2: Understand Partitioned Processing Behavior

Distributed jobs:

- Process partitions independently
- Can succeed partially under transient failures
- Often mark the job complete if errors are not surfaced correctly

Without validation:

- Partial writes remain undetected

Step 3: Conceptual Root Cause

Partial success occurs when:

- Failure handling is weak
- Atomicity is not enforced at partition level
- Completion criteria do not validate data completeness

This creates **silent data quality issues**.

Step 4: Why Obvious Fixes Are Suboptimal

- **Re-run full job:**
  Consumes time and compute, and may duplicate data.
- **Manual data correction:**
  Error-prone, unscalable, and unsafe for financial data.
- **Ignoring missing data:**
  Leads directly to incorrect business decisions.

These approaches prioritize speed over correctness.

Step 5: Validation of Root Cause

To confirm:

- Compare expected vs loaded partition counts
- Reconcile record counts at partition level
- Validate downstream query results

Outcome:
Only specific partitions are missing data.

Step 6: Corrective Actions

- Re-run only the failed partitions
- Ensure idempotent writes
- Add partition-level success validation
- Improve failure alerting for partial loads

This restores correctness efficiently without waste.

Step 7: Result After Fix

- Missing records are restored
- Financial reports are accurate
- SLA is met
- Compute costs are minimize

Final Resolution

- **Root Cause:** Partial partition failure during job execution
- **Fix Applied:** Targeted reprocessing of failed partitions

Key Learnings

- Partial success is more dangerous than full failure
- Partition-level validation is critical in data pipelines
- Efficient recovery targets only what failed
- Senior engineers optimize for correctness and cost

Core Principle Reinforced

Fix only what failed—never rerun what already succeeded.

Scenario 5

Hidden Upstream Schema Change Breaking a Production Pipeline

Problem Statement

A daily ingestion job fails unexpectedly despite no code changes on the consumer side. Investigation reveals that the upstream team deployed a schema change, and multiple downstream tables now depend on this pipeline. The SLA is 45 minutes, and rollback is not an option.

Key Details

- No changes made to ingestion or ETL code
- Upstream schema change deployed independently
- Multiple downstream tables blocked
- SLA risk within 45 minutes
- Upstream rollback not allowed

Clarifying Questions

Before reacting under pressure, a senior engineer asks:

- What exactly changed in the upstream schema?
- Is the change additive, breaking, or incompatible?
- Which downstream transformations are impacted?
- Is the failure deterministic for every run?
- Are schema contracts or versioning in place?

Confirmed Facts & Assumptions

After investigation:

- Upstream added/modified fields without notice
- The change breaks existing ETL assumptions
- Failures occur at schema enforcement / parsing stage
- All dependent downstream tables are affected
- No rollback path exists

Interpretation:

This is a **schema contract violation**, not a runtime or infrastructure issue.

Key Observation

The job did not fail because of data volume or compute limits.
It failed because **data shape changed while the consumer assumed stability**.
This highlights the fragility of pipelines without schema governance.

Root Cause Analysis

Step 1: Inspect Failure Point

- Review error messages and stack traces
- Identify the exact field causing the failure
- Compare previous and current schema versions

Finding:
ETL logic is incompatible with the new schema.

Step 2: Understand Schema Enforcement Behavior

Most ingestion frameworks:

- Enforce strict schemas for correctness
- Fail fast when encountering incompatible changes
- Protect downstream systems from corrupt data

This behavior is intentional.

Step 3: Conceptual Root Cause

Hidden schema changes cause failures when:

- No schema registry or versioning exists
- Producers deploy changes independently
- Consumers assume backward compatibility

This is a **process and contract problem**, not a Spark or SQL problem.

Step 4: Why Obvious Fixes Are Risky

- **Skipping the failing field:**
  Leads to silent data loss and inconsistent analytics.
- **Pausing the job:**
  Guarantees SLA breach without solving the issue.
- **Only contacting upstream:**
  Necessary, but insufficient when SLA is tight.

Senior engineers fix the pipeline first, then fix the process.

Step 5 :Validation of Root Cause

To confirm:

- Diff old vs new schema
- Identify incompatible field changes
- Reproduce failure with updated schema locally

Confirmed:
Upstream schema change broke ETL compatibility.

Step 6:Corrective Actions

- Update ETL logic to handle the new schema safely
- Add backward/forward compatibility where possible
- Implement schema validation at ingestion
- Establish schema contracts and change notifications

This restores functionality while preventing future surprises.

Step 7: Result After Fix

- Pipeline resumes within SLA
- Downstream tables are populated correctly
- No data loss or silent corruption
- Future schema changes are detected early

Final Resolution

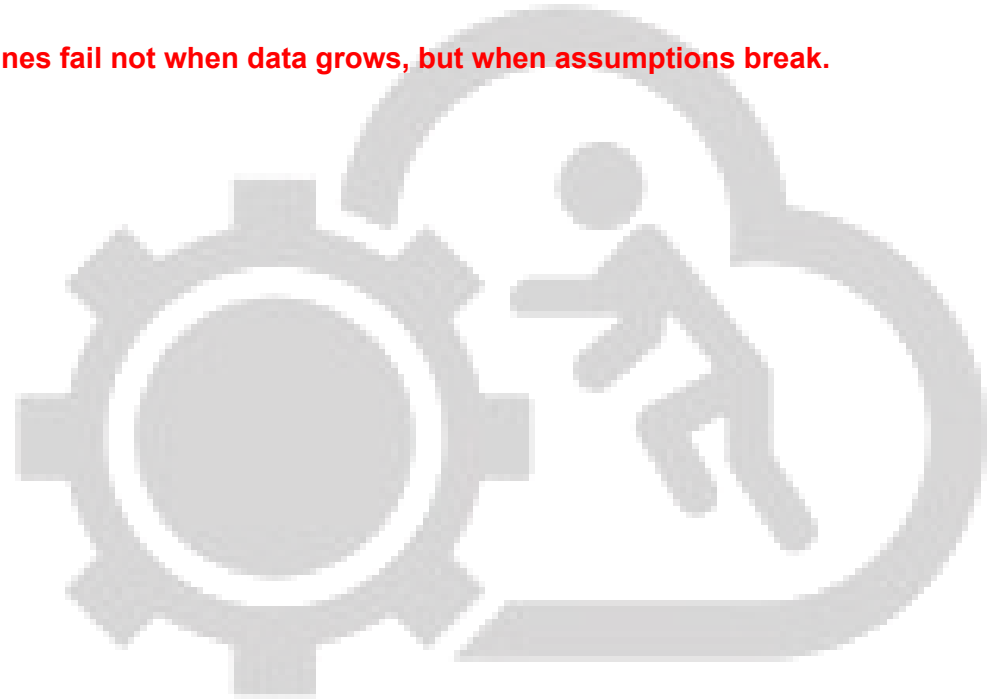
- **Root Cause :** Unannounced upstream schema change
- **Fix Applied :**ETL logic updated with schema compatibility handling

Key Learnings

- Schema changes are inevitable in real systems
- Pipelines must be built to adapt, not assume
- Governance matters as much as code
- Senior engineers design for change, not stability

Core Principle Reinforced

Data pipelines fail not when data grows, but when assumptions break.



Scenario 6

Batch Job Fails Only on Large Input Files

Problem Statement

A production batch ETL job fails only on days when unusually large input files (>50 GB) arrive. No recent code changes were made, the SLA is one hour, and downstream dashboards are business-critical.

Key Details

- Job succeeds with normal-sized inputs
- Fails consistently when very large files arrive
- No recent code or configuration changes
- Downstream dashboards depend on this data
- SLA breach risk within 1 hour

Clarifying Questions

Before reacting, a senior engineer asks:

- Do failures correlate exactly with file size?
- At which stage does the job fail (read, shuffle, write)?
- Is the file a single large object or already partitioned?
- Are failures consistent or intermittent for large files?
- Is memory, shuffle, or executor utilization spiking?

Confirmed Facts & Assumptions

After investigation:

- Failures occur only when files exceed ~50 GB
- Smaller files always process successfully
- Job fails during processing, not scheduling
- Cluster configuration has not changed
- No retries or transient infra issues observed

Interpretation:

The failure is **input-size dependent**, not code-change dependent.

Key Observation

A well-designed batch pipeline should scale with input size.
Failure only on large files indicates:

- Poor input partitioning
- Data skew or single-partition overload
- Memory pressure caused by uneven work distribution
-

This shifts focus from “add more resources” to **how data is read and partitioned**.

Root Cause Analysis

Step 1: Inspect Job Logs and Metrics

- Review failure stack traces
- Check memory and shuffle metrics
- Identify whether one task or executor fails

Finding:
A small number of tasks process disproportionately large data.

Step 2: Understand Large File Processing Behavior

When large files are ingested:

- If not split properly, Spark assigns large chunks to few tasks
- One executor becomes overloaded
- Other executors sit idle

This causes:

- Out-of-memory errors
- Long GC pauses
- Task failures

Step 3: Conceptual Root Cause

The root cause is **improper handling of variable input sizes**:

- Large files are not split into optimal partitions
- Parallelism collapses under skewed workloads
- The system fails under load instead of scaling

This is a design issue, not an infrastructure one.

Step 4 : Why Obvious Fixes Are Incomplete

- **Increasing memory or executors:**
  Provides temporary relief but does not fix poor partitioning.

- **Skipping large files:**
  Causes missing data and invalid dashboards.

- **Only investigating logs without changes**:
  Identifies the issue but doesn't resolve it.

Senior engineers fix how work is distributed, not just how much compute is available.

Step 5 : Validation of Root Cause

To confirm:

- Compare partition sizes for small vs large files
- Identify tasks with extreme input size
  Verify executor utilization imbalance

Outcome:
Large files are processed by too few partitions.

Step 6: Corrective Actions

- Split large files into smaller chunks at ingestion
- Enforce input file size standards
- Repartition data before heavy transformations
- Add checks for abnormal file sizes

This restores parallelism and stability.

Step 7 :Result After Fix

- Large files process successfully
- Executor utilization becomes balanced
- Job completes within SLA
- Downstream dashboards remain reliable

Final Resolution

- **Root Cause:** Improper partitioning of large input files
- **Fix Applied:** File splitting and improved data partitioning

Key Learnings

- Input size variability must be handled by design
- Large files can silently break parallelism
- Scaling hardware is not a substitute for good partitioning
- Senior engineers design pipelines for worst-case inputs

Core Principle Reinforced

Pipelines must scale with data size, not fail because of it.

Scenario 7

Upstream Data Delay Breaking Downstream SLA

Problem Statement

A downstream ETL job fails because its upstream dependency runs two hours late. The overall SLA is three hours, data is only partially available, and the upstream pipeline cannot be forced to rerun

.

Key Details

- Strict end-to-end SLA of 3 hours
- Upstream pipeline delayed by ~2 hours
- Downstream job depends on complete upstream data
- Partial data is available but incomplete
- No control over upstream rerun timing

Clarifying Questions

Before acting, a senior engineer asks:

- Is the downstream job designed to handle late or partial data?
- Are downstream consumers tolerant of delayed delivery?
- Does the pipeline support watermarking or late-arrival handling?
- What is the historical tolerance for SLA breaches?
- Is partial data more dangerous than delayed data?

These questions determine whether the issue is **timing** or **correctness**.

Confirmed Facts & Assumptions

After investigation:

- Upstream delay is genuine and ongoing
- Only partial upstream data is available
- Downstream job expects full upstream completion
- Running downstream early produces incomplete metrics
- Upstream rerun or acceleration is not possible

Interpretation:

This is a **dependency timing issue**, not a system failure.

Key Observation

The pipeline is failing not because of errors, but because:

- Time-based dependencies were violated
- Downstream execution assumes upstream completeness

This exposes a gap between **SLA design and dependency**

reality.

Root Cause Analysis

Step 1: Analyze Dependency Enforcement

- Downstream job starts based on schedule, not readiness
- No guardrail to validate upstream completion

Finding:
The system prioritizes time over data readiness.

Step 2: Understand Partial Data Risk

Processing partial upstream data:

- Produces misleading metrics
- Breaks aggregates and trends
- Creates silent data quality issues

Once consumed, incorrect data is harder to correct than delayed data.

Step 3: Conceptual Root Cause

The root cause is **misaligned dependency handling**:

- Downstream jobs are time-triggered instead of data-triggered
- No late-data or readiness awareness exists
- SLA is defined without accounting for upstream variability

This is a pipeline design issue.

Step 4 :Why Obvious Fixes Are Dangerous

- **Starting downstream anyway:**
  Produces incorrect metrics and erodes data trust.
- **Partial processing:**
  Introduces inconsistency unless explicitly designed for it.
- **Only notifying stakeholders:**
  Transparent but reactive; does not fix the pipeline.

Senior engineers protect correctness even when SLAs are tight.

Step 5 :Validation of Root Cause

To confirm:

- Compare expected vs available upstream partitions
- Validate downstream aggregates on partial data
- Reproduce failure by delaying upstream in staging

Outcome:
Downstream failure is caused by upstream lateness.

Step 6 :Corrective Actions

- Gate downstream execution on upstream completion signals
- Implement data-driven triggers instead of fixed schedules
- Add late-data handling or watermarking where applicable
- Improve SLA definitions to reflect dependency chains
- Notify stakeholders proactively with accurate

timelines This balances correctness with transparency.

Step 7 :Result After Fix

- Downstream jobs run only on complete data
- Incorrect metrics are avoided
- SLA breaches are predictable and explainable
- Pipeline reliability improves over time

Final Resolution

- **Root Cause:** Upstream data delay violating downstream dependency assumptions
- **Fix Applied:** Data-readiness gating and dependency-aware scheduling

Key Learnings

- Time-based triggers are dangerous in dependency-heavy pipelines
- Partial data is often worse than late data
- SLAs must reflect upstream variability
- Senior engineers design for lateness, not ideal timing

Core Principle Reinforced

Data readiness matters more than clock-based execution.

Scenario 8
Third-Party API Schema Change Breaks ETL Pipeline

Problem Statement

A production ETL job that ingests data from a third-party API fails suddenly due to an unannounced schema change in the API response. There is a strict one-hour SLA, no rollback option on the API side, and multiple downstream tables depend on this pipeline.

Key Details

- External third-party API dependency
- Upstream schema changed without notice
- No rollback or control over API provider
- Multiple downstream consumers affected
- SLA breach risk within 1 hour

Clarifying Questions

Before reacting, a senior engineer asks:

- What exactly changed in the API response?
- Is the change additive, breaking, or incompatible?
- Are fields removed, renamed, or re-typed?
- Which downstream tables rely on the affected fields?
- Is schema validation or versioning in place?

Confirmed Facts & Assumptions

After investigation:

- API response schema changed unexpectedly
- ETL job fails during parsing or transformation
- The failure is deterministic for all new API calls
- Downstream tables cannot tolerate missing or malformed fields
- No rollback path exists for the external API

Interpretation:

This is an **external schema contract violation**, not a pipeline infrastructure issue.

Key Observation

Pipelines that depend on external systems are inherently unstable if:

- Schema changes are not validated
- No compatibility layer exists

The job failed correctly by stopping instead of ingesting corrupted data.

Root Cause Analysis

Step 1: Inspect API Response and Failure Point

- Compare previous and current API payloads
- Identify missing, renamed, or type-changed fields
- Map schema changes to ETL transformations

Finding:
ETL logic is incompatible with the updated API schema.

Step 2: Understand External Dependency Behavior

Third-party APIs:

- Can change without notice
- Do not guarantee backward compatibility
- Must be treated as untrusted inputs

Strong validation is mandatory at ingestion boundaries.

Step 3: Conceptual Root Cause

The root cause is **tight coupling to an external schema**:

- No schema validation or adaptation layer
- Assumption of API stability
- Lack of defensive ingestion design

This is a design and governance gap.

Step 4 :Why Obvious Fixes Are Dangerous

- **Skipping the API fetch:**
  Causes data gaps and inconsistent downstream analytics.
- **Pausing downstream pipelines:**
  Guarantees SLA breach without restoring data flow.
- **Only contacting the API provider:**
  Necessary, but insufficient when time-bound SLAs exist.

Senior engineers fix ingestion robustness first, coordination second.

Step 5 :Validation of Root Cause

To confirm:

- Diff old vs new API schema
- Reproduce failure using the new payload
- Validate downstream field dependencies

Outcome:
API schema change is the sole cause of failure.

Step 6 :Corrective Actions

- Adapt ETL logic to handle the new schema safely
- Add schema validation and graceful handling for missing fields
- Introduce versioned schema mapping or abstraction layer
- Implement monitoring and alerts for API contract changes

This restores the pipeline and prevents silent failures.

Step 7 :Result After Fix

- ETL job resumes within SLA
- Downstream tables are populated correctly
- External schema changes are detected early
- Pipeline resilience improves

Final Resolution

- **Root Cause:** Unannounced third-party API schema change
- **Fix Applied:** ETL logic updated with defensive schema handling

Key Learnings

- External systems must be treated as unreliable
- Schema validation is critical at ingestion boundaries
- Pipelines should fail loudly, not ingest silently
- Senior engineers design for change outside their control

Core Principle Reinforced

You cannot control external systems—but you can control how safely you consume them.

Scenario 9
Job Fails Only for Specific Partition Keys

Problem Statement

A daily production job fails consistently for a small subset of partition keys (specific customer IDs), while the rest of the data processes successfully. The SLA is two hours, and an immediate, targeted fix is required.

Key Details

- Failure limited to specific customer IDs
- Majority of partitions succeed
- SLA: 2 hours
- Large production dataset
- Full job rerun is expensive

Clarifying Questions

Before acting, a senior engineer asks:

- Which exact partition keys are failing?
- Is the failure deterministic for those keys?
- Do the failing keys share common data characteristics?
- Is this a data issue or a partitioning issue?
- Can those keys be safely reprocessed independently?

Confirmed Facts & Assumptions

After investigation:

- Failures occur only for a known set of customer IDs
- The same keys fail repeatedly
- Other partitions complete successfully
- No recent code or infrastructure changes
- Data volume is uneven across partitions

Interpretation:

This is a **localized failure**, not a systemic pipeline issue.

Key Observation

When only specific partitions fail:

- The problem is almost always data-specific
- Rerunning the full dataset wastes time and compute
- Precision matters more than brute force

This shifts the strategy from *restart everything* to *isolate and fix*.

Root Cause Analysis

Step 1: Inspect Failing Partitions

- Review logs for failing customer IDs
- Compare record counts and data patterns
- Identify anomalies (nulls, extreme values, malformed records)

Finding:
The issue is isolated to specific partition keys.

Step 2: Understand Partition-Level Processing

Distributed jobs:

- Execute partitions independently
- Fail a job if even one partition fails
- Allow selective reprocessing when designed correctly

This enables targeted recovery.

Step 3: Conceptual Root Cause

Partition-specific failures typically arise from:

- Data anomalies unique to certain keys
- Unexpected edge cases
- Data skew concentrated in a small subset

This is a **data integrity issue**, not a compute capacity issue.

Step 4 : Why Obvious Fixes Are Suboptimal

- **Skipping failing partitions:**
  Leads to missing customer data and incorrect analytics.
- **Restarting the full job:**
  Consumes excessive resources and risks SLA breach.
- **Repartitioning immediately:**
  Useful only if skew is confirmed, not as a first response.

Senior engineers avoid overcorrecting localized problems.

Step 5 : Validation of Root Cause

To confirm:

- Re-run the job only for failing keys
- Validate record counts and transformation logic
- Ensure no duplication or data loss

Outcome:
Targeted reprocessing resolves the issue.

Step 6:Corrective Actions

- Re-run the job only for failing customer IDs
- Add validation checks for partition-specific anomalies
- Improve logging to surface key-level failures early
- Review partitioning strategy if failures repeat

This restores correctness quickly and efficiently.

Step 7: Result After Fix

- Failed partitions are processed successfully
- SLA is met
- Compute costs are minimized
- Full dataset integrity is preserved

Final Resolution

- **Root Cause:** Data anomalies isolated to specific partition keys
- **Fix Applied:** Targeted reprocessing of failing partitions

Key Learnings

- Not all failures require full restarts
- Precision beats brute force in large-scale systems
- Partition-level observability is critical
- Senior engineers fix the smallest failing unit first

Core Principle Reinforced

Fix only what failed—never rerun what already succeeded.

Scenario 10

Intermittent Job Failures Due to Resource Contention

Problem Statement

A critical production pipeline fails intermittently because it competes for resources on a shared cluster. The SLA is one hour, cluster capacity cannot be increased immediately, and other pipelines must continue running.

Key Details

- Shared cluster used by multiple pipelines
- One critical job fails intermittently
- Resource limits are being hit (CPU / memory / slots)
- SLA: 1 hour
- Immediate cluster scaling is not possible

Clarifying Questions

Before reacting, a senior engineer asks:

- Which resources are being exhausted (CPU, memory, executors)?
- Do failures correlate with peak scheduling windows?
- Which other pipelines run concurrently?
- Is the failure deterministic under high load?
- Can job priority or execution windows be adjusted?

Confirmed Facts & Assumptions

After investigation:

- Failures happen only during peak execution windows
- Other non-critical pipelines run concurrently
- Cluster capacity is sufficient outside peak overlap
- No code changes triggered the failures
- Resource exhaustion is transient, not constant

Interpretation:

This is a **scheduling and contention issue**, not a capacity or code issue

Key Observation

The job does not fail because it is inefficient.
It fails because **too many pipelines compete for the same resources at the same time**.
This reframes the problem from *"add more compute"* to *"coordinate execution."*

Root Cause Analysis

Step 1: Analyze Resource Usage Patterns

- Inspect cluster metrics during failure windows
- Identify overlapping pipelines
- Correlate failures with resource saturation

Finding:
Resource contention occurs during scheduling overlap.

Step 2: Understand Shared Cluster Dynamics

In shared clusters:

- Resources are finite
- Jobs compete unless priorities or schedules are enforced
- SLAs conflict when pipelines are not coordinated

This is expected behavior without workload management.

Step 3: Conceptual Root Cause

The root cause is **lack of workload orchestration**:

- No job prioritization
- No scheduling isolation
- Critical and non-critical jobs competing equally

This is an operational design gap.

Step 4 :Why Obvious Fixes Are Suboptimal

- **Increasing cluster size:**
  Solves the problem but doubles cost and avoids addressing scheduling discipline.
- **Optimizing the failing job immediately:**
  Helpful long-term, but not an immediate SLA-safe fix.
- **Ignoring failures:**
  Guarantees SLA breaches and escalations.

Senior engineers first stabilize execution, then optimize.

Step 5: Validation of Root Cause

To confirm:

- Temporarily pause or delay non-critical pipelines
- Observe whether the critical job succeeds
- Monitor resource utilization post-adjustment

Outcome:
Critical job succeeds when contention is removed.

Step 6 :Corrective Actions

- Reschedule non-critical pipelines to off-peak windows
- Introduce job prioritization or resource queues
- Define SLAs at the cluster level, not per job
- Plan medium-term optimization for heavy pipelines

This resolves the issue without increasing cost.

Step 7 :Result After Fix

- Critical pipeline meets SLA consistently
- Other pipelines continue without disruption
- Cluster resources are used predictably
- Operational stability improves

Final Resolution

- **Root Cause:** Resource contention from concurrent pipeline execution
- **Fix Applied:** Rescheduling and prioritization of workloads

Key Learnings

- Shared clusters require workload coordination
- Scheduling is as important as optimization
- Scaling is not always the first or best solution
- Senior engineers balance cost, SLA, and stability

Core Principle Reinforced

In shared systems, scheduling is as important as compute.

Scenario 11

Upstream Data Format Change Breaks ETL Pipeline

Problem Statement

A production ETL job fails because the upstream system changed the input file format from CSV to Parquet. The SLA is one hour, downstream tables are business-critical, and rollback is not possible.

Key Details

- Upstream file format changed without notice
- ETL logic expects CSV input
- Downstream tables are critical for reporting
- SLA: 1 hour
- No rollback option

Clarifying Questions

Before acting, a senior engineer asks:

- Is the format change permanent or temporary?
- Are schemas equivalent between CSV and Parquet?
- Which ingestion layer fails (read, parse, transform)?
- Are multiple formats expected going forward?
- Is format validation present at ingestion?

Confirmed Facts & Assumptions

After investigation:

- Upstream now produces Parquet files only
- ETL job fails at the input parsing stage
- Schema content is logically the same
- Downstream logic remains valid once data is read
- No immediate rollback path exists

Interpretation:

This is an ingestion compatibility issue, not a transformation issue.

Key Observation

File format changes do not change business meaning,
but **tight coupling to a single format makes pipelines brittle**.
The failure is a design limitation, not a data quality issue.

Root Cause Analysis

Step 1: Inspect Failure Point

- Review error logs at read stage
- Confirm parser expects CSV
- Validate Parquet files manually

Finding:
The reader is hard-coded to CSV.

Step 2: Understand Format Differences

- CSV is row-based, schema-on-read
- Parquet is columnar, schema-aware
- Reading logic must be format-aware

Ignoring format evolution leads to ingestion failures.

Step 3: Conceptual Root Cause

The root cause is **rigid ingestion logic:**

- Assumes fixed input format
- No abstraction or detection layer
- No forward-compatibility design

This is a pipeline design gap.

Step 4: Why Obvious Fixes Are Suboptimal

- **Skipping files:**
  Causes missing data and broken downstream metrics.
- **Manual conversion:**
  Slow, unscalable, and risky under SLA pressure.
- **Only notifying business:**
  Necessary, but does not restore data flow.

Senior engineers fix ingestion, not symptoms.

Step 5 :Validation of Root Cause

To confirm:

- Read Parquet files using a test reader
- Validate schema compatibility
- Replay ingestion with updated reader logic

Outcome:
Data processes successfully once the reader is updated.

Step 6 :Corrective Actions

- Update ETL to read Parquet format
- Abstract input format handling
- Add format detection and validation
- Monitor upstream format changes proactively

This restores the pipeline and hardens it for the future.

Step 7 : Result After Fix

- Job completes within SLA
- Downstream tables are populated correctly
- Pipeline supports future format changes
- Operational risk is reduced

Final Resolution

- **Root Cause:** Upstream file format change (CSV → Parquet)
- **Fix Applied:** Updated ingestion logic with format awareness

Key Learnings

- Format changes are operational events, not failures
- Ingestion layers must be flexible
- Hard-coded assumptions create fragile pipelines
- Senior engineers design for evolution

Core Principle Reinforced

Flexible ingestion beats rigid assumptions.

Scenario 12

Production Failure After ETL Code Refactor

Problem Statement

A production ETL job fails after a recent code refactor. The failure occurs only for edge-case data, processes ~1 TB of data, the root cause is not immediately obvious, and the SLA is one hour.

Key Details

- Recent ETL refactor deployed
- Failure triggered by edge-case data
- Large production data volume (~1 TB)
- Root cause unclear under time pressure
- SLA: 1 hour

Clarifying Questions

Before acting, a senior engineer asks:

- Did the refactor change logic or only structure?
- Is the failure deterministic for specific records?
- Are there differences between test and production data?
- Can the issue be safely debugged within the SLA?
- Is rollback operationally safe?

Confirmed Facts & Assumptions

After investigation:

- Failure began immediately after refactor deployment
- Job fails consistently on the same edge cases
- No infrastructure or data source changes occurred
- Debugging would require deep analysis
- A previous stable version exists

Interpretation:

This is a **code regression**, not a data or infrastructure issue.

Key Observation

Under a tight SLA:

- Speed and safety matter more than elegance
- Fixing forward without understanding the bug is risky
- Restoring known-good behavior is the priority

This reframes the goal from *debugging* to *stabilization*.

Root Cause Analysis

Step 1: Identify Change Surface

- Compare refactored code with previous version
- Identify logic touched by the refactor
- Map failure to changed components

Finding:
The refactor altered logic affecting edge-case handling.

Step 2: Understand Refactor Risk

Refactors often:

- Improve readability or structure
- Accidentally change execution semantics
- Fail only under rare data conditions

Edge-case failures are common refactor risks.

Step 3: Conceptual Root Cause

The root cause is **insufficient validation of refactored code on representative data**.
The issue is not that refactoring was done — but that it was not tested at production scale and diversity.

Step 4 : Why Obvious Fixes Are Dangerous

- **Patching in production:**
  Introduces new, untested logic under pressure.
- **Retrying the job:**
  Wastes time; deterministic failures will recur.
- **Ignoring the failure:**
  Guarantees SLA breach and business impact.

Senior engineers favor safe rollback over risky fixes.

Step 5 : Validation of Root Cause

To confirm:

- Roll back to the last stable version
- Re-run the job on the same data
- Verify successful completion

Outcome:
The job succeeds after rollback.

Step 6 : Corrective Actions

- Roll back to the stable code version immediately
- Perform root-cause analysis offline
- Add edge-case data to test suites
- Introduce canary or shadow runs for refactors

This restores service quickly and prevents recurrence.

Step 7 :Result After Fix

- Job completes within SLA
- Downstream systems receive correct data
- Production stability is restored
- Refactor can be safely revisited later

Final Resolution

- **Root Cause:** Code regression introduced during refactor
- **Fix Applied:** Immediate rollback to stable version

Key Learnings

- Rollback is a valid and senior engineering decision
- Refactors require production-like test coverage
- Stability beats elegance under SLA pressure
- Debugging can wait; correctness cannot

Core Principle Reinforced

When in doubt under pressure, restore stability first.

Scenario 13

Pipeline Failure During Holiday / Peak Traffic

Problem Statement

A production job that runs reliably under normal conditions fails during holiday periods when data volume spikes significantly. The SLA is strict, reports are business-critical, and the traffic surge is temporary but predictable.

Key Details

- Job succeeds under normal load
- Fails only during peak / holiday traffic
- Business-critical reporting depends on this job
- Strict SLA with no tolerance for delay
- Data spike is temporary but recurring

Clarifying Questions

Before acting, a senior engineer asks:

- How much does data volume increase during peak days?
- Is the failure due to compute limits, memory pressure, or timeouts?
- Are peaks predictable (calendar-driven)?
- Is auto-scaling available?
- Does the system have defined peak-capacity assumptions?

Confirmed Facts & Assumptions

After investigation:

- Data volume increases sharply during holidays
- Failures occur due to resource exhaustion
- No code or logic changes caused the issue
- Normal-day capacity is insufficient for peak load
- Peak traffic is expected every year

Interpretation:
This is a **capacity planning issue**, not a data or code defect.

Key Observation

Systems that work only on average days are **under-designed.**
Peak load, not average load, determines whether SLAs are met.

Root Cause Analysis

Step 1: Analyze Resource Utilization During Peak

- Review CPU, memory, and executor usage
- Compare normal vs peak runs
- Identify saturation points

Finding:
The cluster hits resource limits only during peak days.

Step 2: Understand Peak Load Dynamics

Holiday traffic:

- Increases input volume
- Extends processing time
- Amplifies any inefficiency

If pipelines are sized only for average load, they will fail under stress.

Step 3: Conceptual Root Cause

The root cause is **insufficient peak-capacity planning**:

- No buffer for predictable spikes
- Static cluster sizing
- Assumption that “normal behavior” is enough

This is a design oversight, not an operational accident.

Step 4 :Why Obvious Fixes Are Dangerous

- **Skipping peak-day processing:**
  Creates gaps in business-critical reports.
- **Reducing data processed:**
  Produces incomplete and misleading analytics.
- **Retrying multiple times:**
  Wastes resources and delays resolution.

Senior engineers design systems to absorb stress, not avoid it.

Step 5 :Validation of Root Cause

To confirm:

- Simulate peak data volumes
- Run job with scaled resources
- Validate SLA compliance

Outcome:
The job succeeds when capacity matches peak load.

Step 6 :Corrective Actions

- Scale the cluster during peak periods
- Enable scheduled or auto-scaling
- Define peak-capacity SLAs explicitly
- Review pipelines for peak-day bottlenecks

This handles spikes without compromising correctness.

Step 7 :Result After Fix

- Job completes successfully during holidays
- SLAs are consistently met
- Business reports remain accurate
- Infrastructure cost is optimized via temporary scaling

Final Resolution

- **Root Cause:** Insufficient capacity for predictable peak traffic
- **Fix Applied:** Temporary cluster scaling for peak periods

Key Learnings

- Average load is misleading; peaks define reliability
- Holiday traffic should never be a surprise
- Capacity planning is a senior responsibility
- Scaling for peaks is cheaper than fixing failures

Core Principle Reinforced

Design for the worst day not the average day.
If you want, next we can do:

Scenario 14

Late-Arriving Data Breaks Daily Pipeline

Problem Statement

A daily pipeline fails because some upstream source tables update later than expected. The SLA is two hours, downstream metrics require complete data, and the upstream pipeline cannot be forced to rerun.

Key Details

- Pipeline depends on previous day's data completeness
- Some sources arrive late on certain days
- Downstream metrics require full data accuracy
- SLA: 2 hours
- No control over upstream execution timing

Clarifying Questions

Before acting, a senior engineer asks:

- Which source tables are arriving late?
- How late do these sources typically arrive?
- Are downstream consumers tolerant of delayed delivery?
- Does the pipeline support late-arriving data handling?
- Is there a readiness signal or watermark mechanism?

Confirmed Facts & Assumptions

After investigation:

- Late-arriving data is a recurring upstream behavior
- The job fails when required source tables are incomplete
- Running early produces incomplete metrics
- Upstream timing cannot be controlled
- Business correctness is more critical than speed

Interpretation:
This is a **data readiness issue**, not a pipeline or infrastructure failure.

Key Observation

The pipeline fails because it assumes **time-based availability**, not **data-based readiness**.
Late-arriving data is normal in distributed systems and must be handled explicitly.

Root Cause Analysis

Step 1: Identify Late Sources

- Compare expected vs actual arrival times
- Identify tables frequently missing at job start
- Measure lateness distribution over time

Finding:
A subset of sources consistently arrives after the scheduled job start.

Step 2: Understand Impact of Early Execution

Running the job without complete data:

- Produces incorrect aggregates
- Breaks downstream metrics
- Requires costly corrections later

Incorrect data is harder to fix than delayed data.

Step 3: Conceptual Root Cause

The root cause is **pipeline design that does not account for late-arriving data**:

- Execution triggered by schedule, not readiness
- No waiting or watermark logic
- SLA defined without upstream variability

This is a design gap, not an operational mistake.

Step 4 : Why Obvious Fixes Are Risky

- **Running the job anyway:**
  Produces incomplete and misleading metrics.
- **Filling missing data with defaults:**
  Introduces silent data corruption.
- **Only notifying business:**
  Transparent, but does not fix the pipeline.

Senior engineers protect correctness before speed.

Step 5 : Validation of Root Cause

To confirm:

- Delay job execution until all sources are available
- Re-run with complete data
- Validate downstream metrics

Outcome:
The job succeeds once late data arrives.

Step 6 : Corrective Actions

- Wait for late-arriving data before execution
- Introduce data-readiness checks or watermarks
- Adjust SLA definitions to reflect upstream variability
- Add monitoring for late-source detection
- Communicate predictable delays proactively

This balances reliability with transparency.

Step 7 : Result After Fix

- Pipeline runs with complete data
- Downstream metrics are accurate
- SLA behavior becomes predictable
- Operational trust improves

Final Resolution

- **Root Cause:** Late-arriving upstream data
- **Fix Applied:** Data-readiness gating before job execution

Key Learnings

- Late-arriving data is normal, not exceptional
- Time-based triggers are fragile in data pipelines
- Correct data late is better than incorrect data on time
- Senior engineers design for data variability

Core Principle Reinforced

Data readiness matters more than schedule adherence.

Scenario 15

Job Fails Due to Configuration Drift

Problem Statement

A daily production ETL job fails after a cluster-level configuration change (for example, Spark version upgrade or memory limit adjustment). The SLA is one hour, rollback is not available, and multiple downstream pipelines are blocked.

Key Details

- No application code changes
- Cluster or Spark configuration recently modified
- Multiple downstream dependencies impacted
- SLA: 1 hour
- Platform rollback not possible

Clarifying Questions

Before acting, a senior engineer asks:

- What configuration changes were applied recently?
- Did Spark defaults or execution behavior change?
- Is the failure deterministic after the upgrade?
- Are other pipelines affected?
- Can job-level configs override platform defaults?

Confirmed Facts & Assumptions

After investigation:

- Job failures started immediately after config change
- Code and data inputs remain unchanged
- Restarting the job does not help
- Reverting configuration would impact other pipelines
- Failure is reproducible

Interpretation:
This is a **platform-induced regression**, not a data or code defect.

Key Observation

Configuration drift can silently invalidate assumptions made by stable pipelines.
Jobs tightly coupled to platform defaults are vulnerable during upgrades.

Root Cause Analysis

Step 1: Identify Configuration Changes

- Review cluster upgrade notes and change logs
- Compare old vs new Spark defaults
- Identify changes impacting memory, shuffle, or execution behavior

Finding:
One or more configuration changes conflict with existing job settings.

Step 2: Understand Platform Evolution

Platform upgrades often:

- Change defaults
- Enforce stricter limits
- Deprecate legacy behavior

Jobs must adapt to remain compatible.

Step 3: Conceptual Root Cause

The root cause is **misalignment between job configuration and updated platform behavior.** This is an operational governance issue, not an implementation bug.

Step 4 : Why Obvious Fixes Are Risky

- **Reverting cluster configuration:**
  Can destabilize other pipelines.
- **Restarting without changes:**
  Deterministic failure will repeat.
- **Only notifying the team:**
  Necessary, but does not restore service.

Senior engineers adapt workloads instead of rolling back shared platforms.

Step 5 : Validation of Root Cause

To confirm:

- Adjust job-level configuration to align with new defaults
- Re-run the job
- Validate downstream recovery

Outcome:
The job succeeds after configuration alignment.

Step 6 : Corrective Actions

- Update job configs for the new platform
- Externalize and version configuration
- Add pre-upgrade compatibility testing
- Monitor and alert on cluster changes

This restores stability and reduces future incidents.

Final Resolution

- **Root Cause:** Configuration drift after cluster upgrade
- **Fix Applied:** Job configuration updated to match new platform behavior

Key Learnings

- Configuration changes can break stable pipelines
- Platform upgrades require workload alignment
- Rollback is not always the safest option
- Senior engineers anticipate and manage drift

Core Principle Reinforced

Stable data pipelines require continuous alignment with evolving platforms.