FABRIC ENGINEERING PATTERNS

Pipelines and Data Factory

Common Patterns. Clear Fixes.

50 engineering patterns covering pipeline design, triggers, copy activity, Dataflow Gen2, error handling, connections, and configuration in Microsoft Fabric Data Factory. Patterns every Fabric engineer must recognise, covering the gap between how Fabric Data Factory is described and how it actually behaves.

Download all books free at ssanjaychandra.com/freedownloads

AUTHOR

Sanjay Chandra

50

SCENARIOS

INTRODUCTION

Pipelines break in different ways.

Book 1 of this series covered Lakehouse and PySpark. The problems there were mostly about how data was stored and how Spark executed against it. This book covers a different layer entirely: how data moves, how pipelines are orchestrated, and how Fabric Data Factory behaves under real production conditions.

Pipeline problems have a different character from Spark problems. They are often harder to diagnose because the error messages are less precise, the execution is more distributed across activities, and the failure modes involve timing, credentials, triggers, and concurrency in ways that do not show up in a Spark UI.

The book is organised into six chapters. Chapter 1 covers pipeline design mistakes that compound over time. Chapter 2 covers triggers and scheduling, where problems often appear intermittently and are hard to reproduce. Chapter 3 covers the copy activity, which is deceptively simple but has many edge cases at scale. Chapter 4 covers Dataflow Gen2, which behaves differently in pipeline context than in standalone execution. Chapter 5 covers error handling and monitoring, where the gap between what Fabric reports and what actually happened is often significant. Chapter 6 covers connections and configuration, where problems frequently only appear in production.

Throughout this book, all scenarios are specific to Microsoft Fabric Data Factory. There are no references to Azure Data Factory concepts like Integration Runtime, Linked Services, or Datasets. Everything here applies to the Fabric-native pipeline experience.

Each scenario follows the same structure as Book 1: the situation, the wrong thinking, what is actually happening, and the fix. Read it cover to cover or use it as a reference when a pipeline breaks in a way you have not seen before.

Reflects the state of Microsoft Fabric as of Q2 2026, after Workspace Identity expansion to data pipelines and notebooks, the F-SKU migration from Premium, and the introduction of Variable Libraries. Behaviors marked as current may change as the platform evolves.

DIAGNOSTIC INDEX

Start with the symptom. Not the chapter.

When a pipeline breaks in production, the symptom is what you have. Find it below, check the scenarios in the order listed, and stop at the one that matches.

Pipeline succeeded but data is wrong or missing.
Check 34, 35, 37, 38, 42 — the "silent success" family. Pipelines lie about success when they detect no error but not expected outcome.

First run of the day is slow, then fast.
Check 02 (Spark cold start). Warm-up pipeline pattern; High Concurrency Mode for notebook-heavy pipelines.

Concurrency makes things slower, not faster.
Check 14 (capacity contention with native queueing), 11 (nested ForEach explosion), 06 (parallelization overhead).

Trigger fires unexpectedly (too many times, wrong day, not at all).
Check 17 (DST skip), 18 (storage event multi-fire), 20 (timezone), 22 (token expiry).

Retries keep failing with the same error.
Check 36 (retry interval too short). Immediate retry against a recovering resource fails consistently. Set the interval to match recovery time.

Connection works interactively but fails on schedule.
Check 43, 45 (token vs service auth), 44 (credential expiry). Modern fix: Workspace Identity instead of personal credentials.

Pipeline gets slower over time despite same volume.
Check 01 (small-file accumulation, log size), 09 (Lookup activity overhead), 19 (queue buildup).

Pipeline times out on large data, succeeds on small.
Check 07 (copy parallelism not enabled), 23 (REST request timeout), 27 (source not partitioned).

Two pipelines writing to same table fail intermittently.
Check 05 (Delta concurrent write conflict). Partition strategy for partitioned tables, serialization for unpartitioned.

Data shifted, truncated, or type-mismatched after copy.
Check 12 (type inference), 24 (null/empty string), 26 (date/timezone), 13 (silent row drops).

Dataflow Gen2 works in preview but breaks in pipeline run.
Check 28, 30, 33. Preview and scheduled runs evaluate differently; folding can break between them.

Pipeline works in dev, fails or behaves differently in prod.
Check 47, 48 (parameter binding), 49 (dynamic content), 50 (cross-workspace connections). Use Variable Libraries for parameterization.

Several scenarios appear under multiple symptoms because one cause produces several symptoms. Check the highest-listed scenario first. If your symptom is not above, the table of contents is organized by cause.

FABRIC ENGINEERING PATTERNS

Chapter 1 Pipeline Design

Fourteen scenarios covering how pipeline architecture decisions create performance and reliability problems that are invisible until production load exposes them.

PIPELINE DESIGN

01 Your incremental load is getting slower with every run despite loading the same volume each time.

S

THE SITUATION

A daily pipeline appends roughly the same number of rows each run. The load took 3 minutes when the table was new. Six months later the same incremental load of the same row count takes 18 minutes. The data volume has not changed.

W

WRONG THINKING

Engineers check the new data for anomalies, review network throughput to OneLake, and look at source system performance. None of these are the cause. The incremental volume is identical so the assumption is that load time should also be identical.

H

WHAT IS ACTUALLY HAPPENING

Two compounding causes, in order of frequency. (1) **Small-file accumulation.** Each incremental write appends one or more new parquet files. After six months of daily appends without compaction, the table has thousands of small files. Before every write, Spark lists all the active files in the table to plan the operation, and that listing cost grows linearly with file count. The write itself is fast; the planning around it is what slows down. (2) **Transaction log size when checkpoints lag.** Spark reads the Delta transaction log to determine current table state. Delta creates checkpoint files periodically to avoid re-reading the entire log — but if checkpoints are not firing as expected, the log read cost grows over time. Checkpoints are designed to bound this cost; when they work, log size is not the bottleneck.

F

THE FIX

Address the file count first because it is the dominant cause. Run `OPTIMIZE` on the table to compact small files into larger ones, then `VACUUM` with appropriate retention to remove obsolete files. Schedule both as recurring maintenance — daily or weekly depending on write frequency. For the log itself, verify checkpoints are firing using `DESCRIBE HISTORY tablename`. Fabric Spark (based on Databricks Runtime 11.1+) creates a checkpoint every 100 commits by default; open-source Delta defaults to 10. If you have a very write-heavy table and want more frequent checkpoints, set `delta.checkpointInterval` via `ALTER TABLE`. Treat Delta table maintenance as part of the pipeline itself, not a separate task. The point is to keep the file count and the log length both bounded — not to choose between them.

PIPELINE DESIGN

02 Your first pipeline run of the day is always slow. Every subsequent run is fast.

S

THE SITUATION

A pipeline that consistently completes in 4 minutes takes 12 to 15 minutes on its first run each morning. Every subsequent run throughout the day completes at the expected speed. Nothing changes between runs. The data volume is identical.

W

WRONG THINKING

Engineers assume overnight batch jobs left the system in a degraded state or that early morning capacity is lower. They schedule the pipeline to run twice and discard the first result, which works but wastes capacity without addressing the root cause.

H

WHAT IS ACTUALLY HAPPENING

Fabric serverless Spark sessions are fully deallocated when idle overnight. The first run of the day starts a cold session, which involves provisioning the session container, loading the Spark runtime, connecting to the Delta catalog, and warming the OneLake file cache. This cold start overhead adds several minutes before the pipeline executes a single activity. Subsequent runs reuse the warm session and populated cache, which is why they run at expected speed.

F

THE FIX

Schedule a lightweight warm-up pipeline to run 5 to 10 minutes before your main pipeline each morning. The warm-up simply reads a small amount of data from the same Lakehouse, which starts the session and begins warming the cache. Your main pipeline then runs against an already-warm session. If your pipeline is notebook-heavy, enable High Concurrency Mode for notebooks in the workspace settings — this packs multiple notebook steps into a single shared Spark session and dramatically reduces per-notebook startup time. Note that High Concurrency Mode applies specifically to notebook activities within a pipeline; it is not a general "keep sessions alive across pipeline runs" feature. For pipelines without notebooks, the warm-up pattern is the practical answer.

PIPELINE DESIGN

03 You switched from full load to incremental. Data started going missing.

S

THE SITUATION

A pipeline was migrated from a full load to an incremental load to improve performance. After the migration, downstream reports start showing gaps. Certain records that exist in the source are missing from the target. The incremental pipeline runs without errors.

W

WRONG THINKING

Engineers check the watermark logic and confirm it looks correct. The missing records are not in any error log. Since the pipeline succeeds, the assumption is that the records were never in the source or were filtered out intentionally.

H

WHAT IS ACTUALLY HAPPENING

The watermark is based on a timestamp column but the source system updates records without changing the timestamp, or the timestamp has lower precision than the update frequency. Records updated between watermark checkpoints are invisible to the incremental logic. Late-arriving records with timestamps earlier than the last watermark are permanently skipped because the incremental window has already moved past them. Full loads caught everything. Incremental loads only catch what the watermark logic is designed to see.

F

THE FIX

Validate that the watermark column is always updated on every record change including soft updates. Add a small overlap window to the incremental range, reprocessing the last 15 to 30 minutes of the previous window on each run, to catch late-arriving records. For sources where update timestamps are unreliable, consider using a change data capture approach or a row hash comparison rather than a pure watermark strategy.

PIPELINE DESIGN

04 Your pipeline succeeded. The downstream table has duplicate rows.

S

THE SITUATION

A daily incremental pipeline completes successfully with no errors. A downstream data quality check flags duplicate rows in the target Delta table. The duplicates correspond to the most recent load date. Previous days have no duplicates.

W

WRONG THINKING

Engineers look for duplicates in the source data first. When the source looks clean they assume the incremental logic has a sudden bug. The pipeline infrastructure itself is rarely the first suspect when a job completes without error.

H

WHAT IS ACTUALLY HAPPENING

The pipeline was retried after a transient failure partway through execution. The first run wrote some data to the target table before failing. The retry ran the full pipeline again from the start without cleaning up what the first run had already written. Since the pipeline uses append mode rather than MERGE or idempotent writes, the rows written in the failed first run were not overwritten but appended again on the retry. The retry strategy is not safe for append-mode writes.

F

THE FIX

Design incremental pipelines to be idempotent. Use MERGE instead of append, or delete the target partition before rewriting it on each run. Add a deduplication step at the start of any notebook reading from a table written by a retryable pipeline. Any pipeline that can be retried must produce the same result whether it runs once or ten times.

PIPELINE DESIGN

05 Two pipelines writing to the same Delta table are producing random failures.

S

THE SITUATION

Two Fabric pipelines both write to the same Delta table as part of separate data flows. Individually both pipelines succeed reliably. When they run at the same time, one or both fail intermittently with a conflict error or a transaction commit failure.

W

WRONG THINKING

Engineers assume Delta Lake handles concurrent writes automatically because it is ACID compliant. They expect the transaction system to resolve conflicts without failures and are surprised when concurrent writes cause errors rather than being silently serialized.

H

WHAT IS ACTUALLY HAPPENING

Delta Lake uses optimistic concurrency control. Both pipelines read the current table version, make their changes, and then attempt to commit. If both commits target overlapping data, Delta detects the conflict and rejects the second commit with a ConcurrentWriteException. Delta does not queue writes automatically. It fails the conflicting one and expects the application to handle the retry. ACID guarantees consistency but does not mean unlimited transparent concurrency.

F

THE FIX

Design the two pipelines to write to non-overlapping partitions so their commits never conflict. If both must write to the same partition, serialize them using pipeline dependencies or scheduling offsets. Add retry logic with exponential backoff to handle ConcurrentWriteException gracefully. Never design two independent pipelines to write to the same Delta partition concurrently without a coordination mechanism.

PIPELINE DESIGN

06 You parallelized your pipeline. It got slower.

S

THE SITUATION

A pipeline processing multiple data sources was refactored to run activities in parallel rather than sequentially. The expectation was a proportional reduction in runtime. Instead, total runtime stayed the same or increased and intermittent failures appeared that did not exist in the sequential version.

W

WRONG THINKING

Parallelism is assumed to always improve throughput. If 5 activities run in sequence taking 10 minutes each, running them in parallel should take 10 minutes total. This is true in theory but ignores how Fabric capacity units are shared across concurrent activities.

H

WHAT IS ACTUALLY HAPPENING

All parallel activities draw from the same Fabric capacity pool simultaneously. When 5 activities that each need significant compute run at the same time, total capacity demand exceeds available units. Fabric throttles activities rather than failing them. Each activity gets a fraction of the compute it would have in isolation, so all 5 run slower. Additionally, concurrent writes to the same target table can cause Delta commit conflicts that were absent in sequential execution.

F

THE FIX

Review workspace capacity utilization during parallel runs using the Fabric Monitoring Hub. If capacity is saturating, reduce parallelism to a level that fits within available capacity units without throttling. Use the ForEach activity batch count setting to control how many iterations run concurrently. For pipelines writing to the same target, serialize the writes even if the reads can be parallelized.

PIPELINE DESIGN

07 Your copy activity copies 10GB without issue. At 100GB it times out silently.

S

THE SITUATION

A copy activity tested successfully on a 10GB dataset. When pointed at the full 100GB production dataset, the activity runs for a long time then fails with a timeout error. No partial data is written and no meaningful error message explains the failure threshold.

W

WRONG THINKING

Engineers assume the timeout is a Fabric platform limit that needs to be increased via a support request. The 10x data increase feels like the obvious cause and increasing the timeout feels like the obvious fix.

H

WHAT IS ACTUALLY HAPPENING

The copy activity is running as a single serial operation. At 10GB this completes within the default timeout window. At 100GB it exceeds the timeout not because the data is too large but because the copy is not partitioned. A single copy thread reading 100GB serially from source takes far longer than the activity timeout allows. The solution is not a longer timeout but a smarter copy strategy that distributes the work.

F

THE FIX

Enable parallel copy in the copy activity settings by configuring the degree of copy parallelism. For database sources, use physical or logical partitioning to split the source into ranges that are read concurrently. For file sources, ensure the source folder contains multiple files so the copy activity can read them in parallel threads. For very large datasets, split the copy into multiple smaller copy activities orchestrated by a ForEach loop with controlled batch size.

PIPELINE DESIGN

08 Your pipeline has 20 activities chained sequentially. Only 3 are actually dependent on each other.

S

THE SITUATION

A pipeline with 20 activities was built sequentially for simplicity. Each activity completes before the next starts. Total runtime is 40 minutes. Business users are asking for a faster pipeline and engineers are unsure where time is being lost.

W

WRONG THINKING

Sequential pipelines feel safe and predictable. Engineers are hesitant to introduce parallelism because it adds complexity. The 40-minute runtime is accepted as the cost of reliability.

H

WHAT IS ACTUALLY HAPPENING

Most of the 20 activities have no logical dependency on each other. They process independent source tables or independent transformation steps. Activity 4 waits for Activity 3 even though it does not use anything Activity 3 produces. This is pure wasted wait time. The pipeline is slow not because the work is slow but because the work is unnecessarily serialized.

F

THE FIX

Map the actual data dependencies between activities. Group activities with no dependencies on each other and connect them in parallel branches. Only enforce sequential ordering where a genuine data dependency exists. In Fabric pipelines, activities without a dependency chain run concurrently by default. Restructuring 20 sequential activities into parallel branches of 3 to 4 genuinely dependent activities can reduce a 40-minute pipeline to under 15 minutes without changing any transformation logic.

PIPELINE DESIGN

09 A pipeline that ran in 10 minutes now takes 45 after adding one Lookup activity.

S

THE SITUATION

A Lookup activity was added to fetch a configuration value from a reference table. The Lookup itself completes in under a second. Yet total pipeline runtime jumped from 10 minutes to 45 minutes after the addition. Nothing else changed.

W

WRONG THINKING

Since the Lookup completes instantly, engineers do not suspect it as the cause. The investigation focuses on downstream activities and transformation logic. The Lookup is dismissed as trivial.

H

WHAT IS ACTUALLY HAPPENING

The Lookup activity is configured to return the first row of a large unfiltered table. To return that first row, Fabric executes a full table scan. If that table is large or unoptimized, the scan takes significant time even though only one row is returned. The Lookup duration in pipeline monitoring shows only the time after the query returns, not the total execution time including the full scan the query engine performed behind the scenes.

F

THE FIX

Always use a filtered query in Lookup activities rather than relying on the first row of the full table. Pass a specific WHERE clause that targets exactly the row you need. If the reference table is queried frequently by multiple pipelines, ensure it is a small optimized Delta table with OPTIMIZE applied. For configuration values that rarely change, consider storing them as pipeline parameters rather than in a large operational table.

PIPELINE DESIGN

10 Your ForEach loop processes 1000 items. It runs them one at a time.

S

THE SITUATION

A ForEach activity iterates over 1000 items, running a copy or notebook activity for each. The pipeline runs for hours even though each individual iteration completes in seconds. Checking the monitoring view shows iterations completing one after another with no overlap.

W

WRONG THINKING

Engineers assume ForEach runs iterations in parallel by default. The sequential behavior is unexpected and is attributed to a Fabric limitation or a configuration option that requires a support request to enable.

H

WHAT IS ACTUALLY HAPPENING

The ForEach activity in Fabric Data Factory runs sequentially by default. Parallel execution must be explicitly enabled by turning off the Sequential setting in the activity properties. When Sequential is on, each iteration waits for the previous one to complete before starting. For 1000 items that each take 5 seconds, sequential execution takes over 80 minutes. With parallel execution and a reasonable batch count, the same 1000 items can complete in a fraction of that time.

F

THE FIX

In the ForEach activity settings, disable the Sequential toggle and set the Batch Count to control how many iterations run concurrently. A batch count of 10 to 20 is a good starting point for most workloads. Do not set batch count too high as each concurrent iteration consumes capacity units. Monitor the Fabric Monitoring Hub during parallel ForEach runs to confirm capacity is not saturating. If iterations write to the same target, ensure they target non-overlapping partitions to avoid Delta commit conflicts.

PIPELINE DESIGN

11 You nested a ForEach inside another ForEach. The pipeline never finishes.

S

THE SITUATION

A pipeline was designed with an outer ForEach iterating over 50 tables and an inner ForEach iterating over 20 partitions per table. With both set to parallel, the pipeline starts but never completes. Activity counts in the monitoring view grow continuously and the pipeline eventually times out or is manually cancelled.

W

WRONG THINKING

The design seems logical. Process each table, and for each table process each partition in parallel. The math looks fine on paper: 50 tables times 20 partitions equals 1000 operations. With parallelism this should complete quickly.

H

WHAT IS ACTUALLY HAPPENING

Nested ForEach with both loops running in parallel creates a combinatorial explosion of concurrent activities. With outer batch count of 10 and inner batch count of 10, up to 100 activities are running simultaneously at peak. Each activity requests capacity units, saturating the workspace pool. Fabric starts queuing activities, which causes more activities to pile up from subsequent iterations. The pipeline enters a state where it is perpetually waiting for queued activities that cannot start because earlier activities are holding all available capacity.

F

THE FIX

When nesting ForEach activities, keep the outer loop parallel and the inner loop sequential, or keep the outer sequential and the inner parallel. Never set both to parallel without calculating the maximum concurrent activity count and verifying it fits within workspace capacity. For 50 outer times 20 inner, a safe approach is outer batch count of 5 with inner sequential, giving a maximum of 5 concurrent outer iterations each running 20 sequential inner steps. Total concurrency stays predictable and capacity saturation is avoided.

PIPELINE DESIGN

12 Your pipeline copies data correctly. The target table has wrong data types.

S

THE SITUATION

A copy activity successfully moves data from a source system to a Lakehouse Delta table. Row counts match. But downstream notebooks fail with type errors, or numeric columns contain string values, or date columns are stored as integers. The copy reported no errors.

W

WRONG THINKING

Since the copy activity succeeded and row counts match, engineers assume the data landed correctly. Type issues are attributed to source data quality problems rather than the copy activity itself. The investigation starts at the source rather than the copy configuration.

H

WHAT IS ACTUALLY HAPPENING

The copy activity infers schema from the source at runtime rather than using an explicit type mapping. When the source returns types that differ from what the target Delta table expects, the copy casts values to strings rather than failing. Numeric columns with occasional null values may be inferred as string. Dates formatted as text strings are copied as-is without conversion. The copy activity prioritizes completing over type correctness when schema is not explicitly defined.

F

THE FIX

Define explicit column mappings in the copy activity with correct target data types for every column rather than relying on schema inference. Pre-create the target Delta table with the correct schema before the first copy run so the copy activity writes into a typed structure. Add a post-copy data quality check that validates key column types using `DESCRIBE TABLE` output before downstream activities consume the data. Never trust schema inference for data pipelines.

PIPELINE DESIGN

13 A copy activity succeeds but 30% of rows are silently dropped.

S

THE SITUATION

A copy activity reports success and the rows written count looks reasonable. But a downstream reconciliation check reveals the target has significantly fewer rows than the source. The discrepancy is consistent across runs. No error or warning appears in the activity output.

W

WRONG THINKING

Engineers check for duplicate rows in the source assuming the reconciliation math is wrong. When source counts are confirmed, the copy activity itself is rarely suspected because it reported success. The missing rows feel like a source system issue.

H

WHAT IS ACTUALLY HAPPENING

The copy activity has fault tolerance settings that allow it to skip incompatible rows rather than failing. When a row cannot be written due to a type mismatch, a null constraint violation, or an encoding issue, the copy silently skips it and counts it as a skipped row rather than a failed row. The activity status is success because the activity itself did not fail. The skipped rows are recorded in the activity output under a separate skipped rows count that is easy to miss when only checking the written rows count.

F

THE FIX

After every copy activity, add a validation step that reads the activity output and checks the skipped rows count. If skipped rows is greater than zero, fail the pipeline explicitly using a Fail activity or set a pipeline variable and check it in a subsequent If Condition activity. Configure the copy activity fault tolerance to fail on incompatible rows rather than skip them for data pipelines where row completeness is critical. Never rely only on the written rows count as a success indicator.

PIPELINE DESIGN

14 You set pipeline concurrency to 10. It runs slower than concurrency 1.

S

THE SITUATION

Pipeline concurrency was increased to allow 10 simultaneous runs of the same pipeline triggered by different events. Individual run times that were 5 minutes at concurrency 1 stretch to 30 minutes or more at concurrency 10. All runs eventually complete but none complete at the expected speed.

W

WRONG THINKING

Higher concurrency should mean more throughput. If one run takes 5 minutes, ten concurrent runs should all complete in roughly 5 minutes in parallel. The slowdown at high concurrency feels like a Fabric bug or a misconfiguration rather than a fundamental resource constraint.

H

WHAT IS ACTUALLY HAPPENING

Ten concurrent pipeline runs are competing for the same workspace capacity units. Each run needs the same Spark session, the same compute, and potentially writes to the same target tables. At concurrency 10, total capacity demand is 10 times what a single run needs. The workspace cannot provision 10x the capacity on demand, so all 10 runs receive throttled resources. Each run takes 6 times longer because it is getting one-sixth of the compute it needs. Concurrency multiplies demand. Capacity does not scale proportionally.

F

THE FIX

Set pipeline concurrency based on available capacity units rather than the number of trigger events. Use the Fabric Monitoring Hub to check peak capacity utilization during concurrent runs and identify the concurrency level at which individual run times remain acceptable. For Spark-based workloads triggered through pipelines, Fabric provides a built-in FIFO queueing mechanism: when concurrent jobs exceed available cores, additional jobs are automatically queued and retried as capacity becomes available (queue expiration is 24 hours; throttled-state capacities reject rather than queue). For event-driven pipelines where burst concurrency causes contention, the practical mitigations are to reduce trigger frequency, batch events before triggering, or use a schedule trigger that fires once per window rather than once per event. Do not attempt to build a queue inside Fabric pipelines using loops and variables — the result is fragile. The native queueing for Spark jobs is sufficient for most cases; reach for external orchestration (Service Bus, Logic Apps) only when you need cross-workspace coordination or stricter ordering guarantees than FIFO.

02

FABRIC ENGINEERING PATTERNS

Chapter 2 Triggers and Scheduling

Seven scenarios covering how Fabric pipeline triggers behave in production and why schedules that look correct on paper produce unexpected execution patterns at scale.

TRIGGERS AND SCHEDULING

15 Your scheduled trigger fires but the pipeline does not run.

S

THE SITUATION

A pipeline trigger is active and shows the correct schedule in the Fabric UI. At the scheduled time, no pipeline run appears in the monitoring history. The trigger shows its last fire time correctly but no corresponding run exists. This happens inconsistently, missing some scheduled runs but not all.

W

WRONG THINKING

Engineers assume a Fabric platform outage or a trigger bug. They disable and re-enable the trigger hoping that resets it. Sometimes this appears to work, reinforcing the belief that the trigger itself is unstable rather than investigating what happens between the trigger firing and the run starting.

H

WHAT IS ACTUALLY HAPPENING

The trigger fired correctly but the pipeline run was rejected because a previous run was still in progress and pipeline concurrency was set to 1. When the trigger fires while the previous run has not yet completed, Fabric does not queue the new run by default. It drops it. The trigger fire is recorded but no run is created. This is the expected behavior for the default concurrency setting, but it is not visible in standard monitoring without checking trigger fire logs separately from pipeline run logs.

F

THE FIX

Increase pipeline concurrency if overlapping runs are acceptable for your use case. If runs must not overlap, add alerting when a scheduled run is dropped by checking trigger fire count against pipeline run count. Reduce the scheduled interval or optimize the pipeline to ensure it always completes before the next trigger fires. For pipelines where missing a run is critical, implement a compensating mechanism that detects and reruns missed executions as part of the next scheduled run.

TRIGGERS AND SCHEDULING

16 Two triggers fire at the same time. One pipeline run overwrites the other.

S

THE SITUATION

Two separate triggers are configured for the same pipeline: one on a schedule and one on a storage event. On days when a file arrives exactly at the scheduled time, one pipeline run produces correct output while the other produces empty or partial results. Which one wins appears random.

W

WRONG THINKING

Engineers investigate the pipeline logic for a race condition bug. Since the pipeline is stateless and the runs look identical in the monitoring view, the random winner pattern feels like a nondeterministic platform bug rather than a design problem.

H

WHAT IS ACTUALLY HAPPENING

Both runs start nearly simultaneously and both attempt to write to the same target partition. The run that starts a few seconds later reads the same source data, performs the same transformation, and then overwrites the output of the first run. Depending on timing, the overwrite may capture a partially written state from the first run. Delta Lake ensures neither run corrupts the table, but the last committed write wins and any work done by the earlier run is replaced entirely.

F

THE FIX

Design triggers so they are mutually exclusive. If a storage event trigger covers the real-time case, the schedule trigger should only fire when no recent event-triggered run has completed successfully. Use a Set Variable activity at the start of each run to record a run timestamp in a control table, and add an If Condition check at the start of the pipeline that exits early if a recent successful run already exists for the same time window. Never rely on two triggers for the same pipeline writing to the same target without a coordination check.

TRIGGERS AND SCHEDULING

17 Your schedule trigger runs at midnight. It skips some days randomly.

S

THE SITUATION

A daily pipeline is scheduled to run at midnight. Most days it fires on time. But on some days there is no run in the monitoring history at all for that day. No error is logged. The trigger shows as active. The skipped days appear random with no obvious pattern.

W

WRONG THINKING

Engineers assume a Fabric platform reliability issue or a time zone configuration problem. They check the trigger time zone setting, confirm it looks correct, and file a support ticket. The skipped days continue occurring after the ticket is closed.

H

WHAT IS ACTUALLY HAPPENING

The trigger is configured with UTC time but the workspace or the engineer reviewing logs is using a local time zone. What appears as a missing midnight run is actually a run that fired at midnight UTC, which corresponds to a different local time. On days when daylight saving time shifts occur, a run that appears to skip a day in local time actually fired at the correct UTC time. Additionally, if the previous day's run ran long and was still active at midnight, the trigger may have dropped the new run due to concurrency limits.

F

THE FIX

Always verify trigger fire times in UTC in the raw monitoring logs rather than relying on local time display in the Fabric UI. Set the trigger time zone explicitly in the trigger configuration to match your intended execution timezone and avoid relying on UTC defaults if your team works across time zones. Add a daily reconciliation check that compares expected run count against actual run count for the previous 7 days and alerts when a gap is detected, regardless of whether the trigger reports firing.

TRIGGERS AND SCHEDULING

18 A storage event trigger fires 50 times for one file drop.

S

THE SITUATION

A storage event trigger is configured to fire when a file is created in a specific OneLake path. When a source system drops one file, the trigger fires dozens of times within seconds and dozens of pipeline runs start simultaneously. The target table ends up with duplicate data and the workspace capacity spikes unexpectedly.

W

WRONG THINKING

Engineers assume the source system is creating multiple files or that the trigger has a bug causing it to fire repeatedly. The focus goes to the source system team to investigate why multiple files are being created when only one was expected.

H

WHAT IS ACTUALLY HAPPENING

The storage event trigger path is configured too broadly. A path like a top-level folder without a specific file pattern fires for every file system event in that path including folder creation, metadata updates, temp file writes, and partial writes that the source system makes before the final file is complete. Each intermediate write triggers a separate event and a separate pipeline run. The source dropped one final file but created many intermediate file system events in the process.

F

THE FIX

Configure the storage event trigger with a specific blob name prefix and suffix that matches only the final file pattern, for example a specific extension like .parquet or .csv rather than any file in the folder. Add a blob name filter that excludes temp files and partial writes by naming convention. Set pipeline concurrency to 1 on event-triggered pipelines to prevent duplicate processing even if multiple events fire. For critical pipelines, add an idempotency check at the start that skips processing if the file has already been successfully processed.

TRIGGERS AND SCHEDULING

19 Your pipeline is scheduled every 15 minutes. It starts queuing after a few hours.

S

THE SITUATION

A pipeline scheduled to run every 15 minutes works correctly for the first few hours of the day. By mid-morning, runs start taking 20 or 25 minutes each. By afternoon, runs are queuing and the pipeline is consistently behind schedule, with the backlog growing throughout the day.

W

WRONG THINKING

Engineers assume the data volume grows throughout the day causing later runs to take longer. They investigate source row counts and confirm they are similar across all runs. The volume explanation does not hold but no other cause is identified.

H

WHAT IS ACTUALLY HAPPENING

The pipeline runs fine at 12 to 14 minutes each when the workspace is quiet. As other workloads start throughout the morning, shared workspace capacity becomes constrained. Each pipeline run now competes for capacity with reports refreshing, notebooks running, and other pipelines executing. Individual run times creep above 15 minutes. The next trigger fires before the current run finishes. With concurrency set to allow multiple runs, queuing begins. Once the backlog starts it compounds: more queued runs means more contention means slower individual runs means more queuing.

F

THE FIX

Measure the 95th percentile run time across the full day, not just early morning runs. Set the schedule interval to be at least 20 to 25 percent longer than the 95th percentile run time to provide headroom during peak capacity hours. Optimize the pipeline to reduce run time below the 15-minute threshold under all capacity conditions. If 15-minute freshness is a hard business requirement, work with workspace administrators to ensure dedicated capacity is available during peak hours or move to a capacity tier that supports the concurrent workload profile.

TRIGGERS AND SCHEDULING

20 You have a daily trigger. It fires twice on some days.

S

THE SITUATION

A pipeline scheduled to run once daily shows two runs in the monitoring history on certain days. Both runs complete successfully. The target table has duplicated data for those days. The trigger configuration shows a single daily schedule with no obvious explanation for the double fire.

W

WRONG THINKING

Engineers check whether someone manually triggered a run on those days. When no manual trigger is found, the duplicate fire is attributed to a Fabric platform bug and a support ticket is raised. The investigation focuses on the trigger mechanism rather than the pipeline configuration history.

H

WHAT IS ACTUALLY HAPPENING

The pipeline has two active triggers: the current one and an older one that was not deactivated when the schedule was updated. Both triggers are pointed at the same pipeline and both fire at their respective scheduled times. If the old trigger fired at 11pm and the new one fires at midnight, both show as separate runs on the same calendar day in local time. The Fabric trigger management UI can show multiple triggers per pipeline but this is easy to overlook when only checking the most recently modified trigger.

F

THE FIX

Audit all triggers associated with a pipeline, not just the most recently modified one. In Fabric, navigate to the pipeline and review all associated triggers from the trigger management panel. Deactivate or delete any triggers that are no longer needed before creating replacement triggers. Establish a naming convention for triggers that includes the pipeline name and a version or date suffix so stale triggers are easy to identify. Add a daily reconciliation check that alerts when run count exceeds expected count for any pipeline.

TRIGGERS AND SCHEDULING

21 Your pipeline depends on another pipeline finishing first. It runs before it.

S

THE SITUATION

Two pipelines are designed with a dependency: Pipeline B should only run after Pipeline A completes successfully. But Pipeline B consistently starts before Pipeline A finishes, producing incorrect results because it reads data that Pipeline A has not yet written.

W

WRONG THINKING

Engineers add a Wait activity at the start of Pipeline B with a fixed duration, estimating how long Pipeline A takes. This works most days but fails when Pipeline A runs longer than usual, and wastes time on days when Pipeline A finishes early.

H

WHAT IS ACTUALLY HAPPENING

The pipelines are not actually connected through a dependency chain. They are two separate pipelines each with their own triggers firing at similar times. There is no Fabric-native mechanism enforcing the order unless Pipeline A explicitly triggers Pipeline B upon successful completion or Pipeline B checks for Pipeline A completion at runtime. Separate triggers with similar schedules do not create a dependency. They create a race.

F

THE FIX

Remove the independent trigger from Pipeline B. Instead, add an Execute Pipeline activity at the end of Pipeline A that calls Pipeline B only on successful completion. This creates a hard dependency enforced by the pipeline execution chain itself. If Pipeline A fails, Pipeline B never starts. This is more reliable than time-based waits and eliminates the race condition entirely.

TRIGGERS AND SCHEDULING

22 A trigger that worked for months suddenly stops firing.

S

THE SITUATION

A pipeline trigger that has been running reliably for several months stops firing without any configuration change. The trigger shows as active in the Fabric UI. No error is logged. The pipeline simply stops appearing in the run history from a specific date.

W

WRONG THINKING

Engineers check for Fabric service incidents on the date the trigger stopped, find nothing, and assume a silent platform change caused the issue. They recreate the trigger which fixes it temporarily but the root cause remains unknown.

H

WHAT IS ACTUALLY HAPPENING

The trigger was created by a user account that has since been deactivated or had its workspace permissions revoked. In Fabric, triggers run in the context of the user or service principal that created them. When that identity loses access to the workspace or the underlying data sources, the trigger continues to show as active in the UI but silently fails to execute any runs. No error surface is shown to workspace administrators because the failure occurs at the identity authorization level before a run is created.

F

THE FIX

Audit who created each trigger in your workspace. Recreate critical triggers using a service principal or a shared service account that will not be deactivated when individual team members leave or change roles. Assign workspace roles to the service principal rather than to individual user accounts for all production pipeline identities. Implement a daily check that confirms expected pipeline run counts and alerts when a pipeline has not run within its expected window.

03

FABRIC ENGINEERING PATTERNS

Chapter 3 Copy Activity

Five scenarios covering how the copy activity behaves at scale and why a successful copy does not always mean correct data landed in the target.

COPY ACTIVITY

23 Your copy activity from a REST API fails after about 100 seconds every time.

S

THE SITUATION

A copy activity pulling data from a REST API source fails consistently at around 100 seconds into execution. The error message references a connection timeout or a request timeout. Smaller data pulls from the same API succeed. The timeout is precise enough to suggest a hard limit rather than a network issue.

W

WRONG THINKING

Engineers contact the API team assuming the API itself has a timeout that needs to be extended. The API team confirms no such limit exists on their side. The investigation stalls because the timeout appears to be coming from somewhere in the middle of the connection chain.

H

WHAT IS ACTUALLY HAPPENING

The default HTTP request timeout in Fabric copy activity for REST sources is `00:01:40` — 100 seconds — per Microsoft's documentation. For REST APIs that return large paginated datasets, the copy activity makes individual HTTP requests for each page. If any single page request takes longer than this default to return, the entire copy activity fails. This is not an API timeout; it is the copy activity client-side timeout per individual request. It is particularly common when requesting large page sizes from APIs that are slow to assemble responses.

F

THE FIX

Reduce the page size in the REST source pagination settings so each individual request returns faster and stays within the 100-second window. Enable pagination explicitly in the copy activity REST source configuration so the activity pages through results rather than requesting everything in one call. If the API supports parallel requests for different date ranges or entity IDs, use a ForEach loop to split the load across multiple smaller copy activities running concurrently rather than one large sequential pull. If you genuinely need a longer per-request timeout, the `requestTimeout` property on the REST source can be increased — but reducing page size is the more reliable fix.

COPY ACTIVITY

24 You copy from source to Lakehouse. Nulls are becoming empty strings.

S

THE SITUATION

A copy activity moves data from a relational source to a Delta table in the Lakehouse. After the copy, columns that had NULL values in the source contain empty strings in the target. Downstream aggregations and null checks produce wrong results because empty string is treated as a valid value, not as null.

W

WRONG THINKING

Engineers assume the source system is sending empty strings rather than nulls and work with the source team to fix the data. After the source fix, the problem persists. The copy activity is not suspected because it reported no data quality issues.

H

WHAT IS ACTUALLY HAPPENING

The copy activity is performing implicit type casting during the write. When the target Delta table schema expects a string column and the source sends a null, some connector configurations convert null to an empty string to avoid a null write into a typed column. This behavior depends on the source connector type and the null handling settings in the copy activity. It is not visible in the activity output because no rows are rejected and no warnings are emitted. The data lands but with silently altered semantics.

F

THE FIX

Review the copy activity column mapping settings and check whether null handling is configured explicitly. For connectors that default to converting nulls, set the null value to remain as null rather than converting to an empty string. Add a post-copy data quality check that counts rows where key nullable columns are empty string and alerts if the count is greater than zero. Pre-create the target Delta table with explicit nullable column definitions so the copy activity writes into a schema that accepts null values correctly.

COPY ACTIVITY

25 Your incremental copy using a watermark is missing rows on every run.

S

THE SITUATION

An incremental copy activity uses a watermark column to fetch only new or updated rows since the last run. Row counts per run look reasonable but a periodic full reconciliation reveals a small but consistent gap. Certain rows that exist in the source never appear in the target.

W

WRONG THINKING

Engineers check the watermark logic and the query generated by the copy activity. Both look correct. The missing rows appear to be legitimate source records that were written after the watermark was captured but before the copy executed. The explanation feels like an edge case that is hard to reproduce consistently.

H

WHAT IS ACTUALLY HAPPENING

The watermark boundary creates a timing gap. The copy activity reads the current watermark value at the start of the run, then queries the source for rows where the timestamp is greater than that value. But between the moment the watermark is read and the moment the source query executes, new rows can be written to the source with timestamps that fall within the intended window but are missed because the query has already been formed. This is the classic watermark boundary race condition. It affects every watermark-based incremental pattern to some degree.

F

THE FIX

Apply an overlap window by subtracting a small buffer from the lower watermark boundary, typically 5 to 15 minutes depending on source write frequency. This causes each run to reprocess a small tail of the previous window, capturing any rows that fell in the race condition gap. Use MERGE rather than append in the target to handle the duplicate processing of the overlapping window without creating duplicate rows. Update the watermark to the start time of the current run rather than the end time to ensure the next run picks up from a clean boundary.

COPY ACTIVITY

26 Copy activity succeeds but date columns are shifted by one day.

S

THE SITUATION

A copy activity moves data from a source system to a Lakehouse table. All columns look correct except date columns, which are consistently one day behind the source values. A source value of 2024-03-15 arrives in the target as 2024-03-14. The shift is consistent across all date values and all runs.

W

WRONG THINKING

Engineers assume the source system is storing dates incorrectly and ask the source team to investigate. When the source team confirms their dates are correct, the investigation moves to the network layer and connection settings before eventually reaching the copy activity configuration.

H

WHAT IS ACTUALLY HAPPENING

Two common causes, both about timezone interacting with type. (1) **UTC-to-local conversion.** The copy activity converts a datetime stored as UTC in the source to a local timezone. `2024-03-15 00:30:00 UTC` converted to UTC minus 1 becomes `2024-03-14 23:30:00`; if the target is a `DATE` column the time portion is dropped and `2024-03-14` remains. (2) **Delta type mapping in Lakehouse sinks.** When the source column is `DATETIME` or `DATETIME2` and the copy activity writes to a Lakehouse Delta column inferred as `DATE`, the time portion is truncated. If any of those datetimes were close to midnight UTC, the truncation can occur after a timezone conversion has already shifted the date. The Fabric-specific path (SQL source → Lakehouse Delta sink with type inference) hits this combination more often than classic Azure Data Factory paths because Delta column types are inferred at first write and stick.

F

THE FIX

Set the copy activity to treat all datetime and date columns as UTC throughout the pipeline without performing any timezone conversion. For Lakehouse Delta sinks, explicitly write the target column as `TIMESTAMP` rather than `DATE` when the source has time information you care about — once a Delta column type is established, it sticks for the life of the table unless you rewrite it. If the source stores dates in local time, document this explicitly and handle the conversion in a downstream transformation notebook rather than in the copy activity. Add a post-copy check that compares a sample of date values between source and target on every run. Never rely on implicit timezone conversion in copy activities for date-sensitive data.

COPY ACTIVITY

27 Your copy activity parallelism is set to 32. It still reads from the source using 1 thread.

S

THE SITUATION

The degree of copy parallelism is configured to 32 in the copy activity settings expecting fast parallel reads from a large source table. The copy runs at the same speed as when parallelism was set to 1. Checking the activity output shows only 1 read thread was used despite the configuration.

W

WRONG THINKING

Engineers assume the parallelism setting is not being respected by Fabric and raise a support request. The support team confirms the setting is valid. The investigation does not progress because the configuration appears correct.

H

WHAT IS ACTUALLY HAPPENING

The parallelism setting controls the maximum number of parallel read threads but the actual number used depends on the source being partitionable. For database sources, parallel reads require the source to be configured with a partition column and a partition range so the copy activity can split the read into independent chunks. Without a partition column defined, the copy activity cannot split the source query into parallel ranges and falls back to a single serial read regardless of the parallelism setting. The setting is valid but the precondition for using it is not met.

F

THE FIX

In the copy activity source settings, configure a partition column that has a well-distributed numeric or date range. Set the partition upper and lower bounds to the actual data range of that column. The copy activity will then split the range into the configured number of parallel chunks and read them concurrently. For tables without a suitable partition column, consider adding a modulo-based partition on a surrogate key. Verify parallelism is working by checking the copy activity output for the actual parallel read count after the first run.

04

FABRIC ENGINEERING PATTERNS

Chapter 4 Dataflow Gen2

Six scenarios covering how Dataflow Gen2 behaves differently in pipeline context than in standalone execution and what causes pipeline runs to diverge from development previews.

DATAFLOW GEN2

28 Your Dataflow Gen2 transformation is correct in preview. Wrong in pipeline run.

S

THE SITUATION

A Dataflow Gen2 transformation produces correct results in the dataflow editor preview. When the same dataflow runs as part of a pipeline, the output contains different values. The transformation logic has not changed between preview and pipeline execution.

W

WRONG THINKING

Engineers re-run the preview multiple times, confirm it is correct, and assume the pipeline is calling a different version of the dataflow. They check version history and find no differences. The inconsistency between preview and pipeline run feels like a platform bug.

H

WHAT IS ACTUALLY HAPPENING

Dataflow Gen2 preview in the editor runs against a sample of the data, typically the first 1000 rows. The full pipeline execution runs against the entire dataset. Transformations that work correctly on a small sample can produce different results on full data for several reasons: aggregations behave differently when all groups are present, joins that were unique on the sample have duplicates on full data, and filter conditions that appeared to work may have edge cases only present in the full dataset. The preview is a design tool, not a correctness test.

F

THE FIX

Never use the editor preview as validation for production logic. After developing the transformation, run the full dataflow in pipeline context against a complete representative dataset and validate the output row count and key column distributions. Add explicit data quality assertions after the dataflow activity in the pipeline that check expected row counts, null rates in key columns, and value range bounds. Treat any discrepancy between preview and pipeline output as a signal to investigate the full dataset for edge cases the sample did not contain.

DATAFLOW GEN2

29 A Dataflow Gen2 that works on 1 million rows crashes on 10 million with no useful error.

S

THE SITUATION

A Dataflow Gen2 transformation processes 1 million rows successfully in testing. When the source grows to 10 million rows in production, the dataflow fails. The error message in the pipeline monitoring is generic, referencing an internal engine error or a memory-related failure with no actionable detail.

W

WRONG THINKING

Engineers try increasing the dataflow compute size expecting that more memory will resolve the issue. After upgrading, the dataflow still fails. The error message remains unhelpful. The investigation stalls because there is no specific component in the dataflow pointed to as the cause.

H

WHAT IS ACTUALLY HAPPENING

Dataflow Gen2 uses a distributed execution engine but certain transformations do not distribute well at scale. Operations like self-joins, complex window functions, full outer joins on high-cardinality keys, and multi-level nested groupings can require the engine to hold large intermediate datasets in memory during execution. At 1 million rows these fit within the engine memory budget. At 10 million rows they exceed it. The generic error message reflects an engine-level failure that does not map cleanly to a specific transformation step, making diagnosis difficult without understanding which transformation is the memory bottleneck.

F

THE FIX

Enable dataflow diagnostics and review the execution plan to identify which transformation step has the largest intermediate output. Simplify the most complex transformations by breaking them into multiple sequential dataflow steps rather than one monolithic transformation. For large joins, filter both sides of the join to the smallest possible dataset before joining. If the dataflow must handle growing data volumes over time, test at 3x to 5x current production volume before declaring the design production-ready. Consider moving the most memory-intensive transformations to a Spark notebook where memory management is more explicit and configurable.

DATAFLOW GEN2

30 Your Dataflow Gen2 debug run shows correct results. The scheduled production run differs.

S

THE SITUATION

A Dataflow Gen2 is validated using the debug run feature which shows correct output. When the same dataflow runs on schedule in production, the output differs. Specific rows are missing or have different values. The debug run continues to show correct results when run manually after the production discrepancy is discovered.

W

WRONG THINKING

Engineers assume the production run is using a cached or stale version of the dataflow and publish it again. The next scheduled run still differs from the debug output. The investigation focuses on the dataflow logic rather than the execution environment.

H

WHAT IS ACTUALLY HAPPENING

Dataflow Gen2 debug runs execute in the context of the currently signed-in user using their personal credentials and can connect to data sources that the scheduled run service identity cannot. The debug run may read from a data source using the engineer's credentials which have access to a different environment or a different permission scope than the service principal used for scheduled runs. Additionally, debug runs can read from local preview caches while scheduled runs always read from the live source. If the source data changed between the debug run and the scheduled run, the outputs will differ for legitimate reasons.

F

THE FIX

Always run the dataflow as a pipeline activity rather than relying on debug runs to validate production behavior. Ensure the service principal or identity used for scheduled pipeline runs has identical access permissions to all data sources that the dataflow connects to. Test the pipeline run explicitly before promoting to a scheduled trigger. Add row count and key column validation steps after the dataflow activity in the pipeline to detect output differences from expected values automatically.

DATAFLOW GEN2

31 You added a derived column in Dataflow Gen2. The whole pipeline slowed dramatically.

S

THE SITUATION

A derived column transformation was added to an existing Dataflow Gen2 to compute a new field. The transformation itself is a simple string concatenation taking milliseconds in preview. After publishing, the pipeline that runs the dataflow takes substantially longer than before the column was added — sometimes many times longer.

W

WRONG THINKING

Engineers review the derived column expression and confirm it is correct and simple. They assume the slowdown is caused by Fabric republishing the dataflow which sometimes takes longer. After waiting a few runs the slowdown persists. The expression is not suspected because it is trivially simple.

H

WHAT IS ACTUALLY HAPPENING

Three things commonly cause this, none of them the derived column expression itself. (1) **Query folding broken.** Power Query attempts to push transformations back to the source database as a single SQL query (folding). A derived column added with an expression that cannot be folded (custom M function, certain string operations, conditional logic) forces the dataflow to load the full unfolded dataset into the Power Query engine for processing. The downstream steps that previously folded now also run in-engine. (2) **Staging behavior changed.** Dataflow Gen2 stages intermediate results to enable downstream consumption. A new step can change where staging happens in the chain, causing data to be re-read or re-materialized. (3) **Schema inference re-runs.** Adding a column can trigger re-evaluation of column types throughout the chain, particularly if the new column is referenced by later steps.

F

THE FIX

In the Power Query Online editor, right-click the new derived column step and check whether "View Native Query" is enabled. If it is greyed out, query folding is broken at that step — that's almost certainly the cause. Rewrite the expression using foldable operations (basic column references, arithmetic, well-supported string functions) instead of custom M, and verify folding is restored. If the derived value cannot be expressed in a foldable way, compute it at the source (a view in the source database) or in a downstream notebook after the dataflow has landed the data in a Lakehouse, rather than inside the dataflow itself. For complex dataflows, split the transformation into multiple dataflow activities so each has a smaller execution plan; the folding boundary is then clearer per stage.

DATAFLOW GEN2

32 Your Dataflow Gen2 join produces more rows than expected in production.

S

THE SITUATION

A Dataflow Gen2 join between two tables produces the correct row count in the editor preview. In the scheduled production run against full data, the output has significantly more rows than the source tables combined. No error is raised. The excess rows contain duplicated values from one side of the join.

W

WRONG THINKING

Engineers check the join condition and confirm it looks correct. They assume the source data has unexpected duplicates and ask the source team to investigate. The source team confirms the data is clean. The join condition continues to look correct in every review.

H

WHAT IS ACTUALLY HAPPENING

The join key has duplicate values in one or both source tables in the full production dataset that were not present in the preview sample. A join in Dataflow Gen2, like any relational join, produces one output row for every matching pair between the two sides. If the left side has 3 rows with key A and the right side has 2 rows with key A, the join produces 6 rows for key A. With small sample data this multiplier is invisible. With full production data containing high-frequency keys, the row explosion is significant. The join is working correctly. The data distribution assumption was wrong.

F

THE FIX

Before any join in a dataflow, validate key uniqueness on both sides using a grouping and count transformation to identify keys with more than one row. Decide whether to deduplicate before joining or whether the duplicates are meaningful and the join type needs to change. Add a row count assertion after the join activity that fails the pipeline if output rows exceed expected count by more than a defined threshold. Never assume key uniqueness in production data without an explicit check.

DATAFLOW GEN2

33 Your Dataflow Gen2 refresh fails when triggered from a pipeline but works when run standalone.

S

THE SITUATION

A Dataflow Gen2 runs successfully when refreshed manually from the Fabric workspace. When the same dataflow is triggered from a pipeline using the Dataflow activity, it fails. The error in the pipeline monitoring is vague. Running the dataflow manually again after the pipeline failure succeeds immediately.

W

WRONG THINKING

Engineers assume the pipeline is calling the wrong dataflow version or that a timing issue causes the failure. They add a Wait activity before the Dataflow activity hoping a delay resolves it. The failure continues. The manual run continuing to succeed makes the pipeline trigger feel like the specific cause.

H

WHAT IS ACTUALLY HAPPENING

The Dataflow activity in a pipeline runs the dataflow using the pipeline identity, which is typically a service principal. The manual refresh runs using the signed-in user identity. If the dataflow connects to a data source that the service principal does not have permission to access, the pipeline-triggered run fails while the manual run succeeds. The error message from the Dataflow activity often does not clearly indicate a permissions failure, instead surfacing a generic refresh error that points at the dataflow engine rather than the identity issue.

F

THE FIX

Check the data source connections used by the dataflow and verify that the service principal or managed identity used by the pipeline has equivalent access permissions to all sources. Grant the pipeline identity the same access as the user identity that runs the manual refresh. For each connection used in the dataflow, verify it is configured to use a service principal credential rather than a personal user credential, which will fail when the pipeline runs outside the context of that user.

05

FABRIC ENGINEERING PATTERNS

Chapter 5 Error Handling and Monitoring

Ten scenarios covering how Fabric pipeline monitoring can mislead you and why the gap between what is reported and what actually happened is wider than you expect.

ERROR HANDLING AND MONITORING

34 Your pipeline failed. The error message says success.

S

THE SITUATION

A pipeline run shows a green success status in the Fabric monitoring view. But the target Delta table is empty or contains stale data. Downstream reports are wrong. When engineers investigate, the pipeline genuinely ran and all activities show success status. No error is visible anywhere in the monitoring UI.

W

WRONG THINKING

Engineers assume a data issue upstream and contact source system teams. When source data is confirmed correct, the investigation circles without finding a cause because the pipeline monitoring consistently shows success. The possibility that a succeeded pipeline can produce wrong output is not considered.

H

WHAT IS ACTUALLY HAPPENING

The pipeline has an activity failure somewhere in the execution path but the activity dependency conditions are configured to continue on failure rather than stop. A copy activity that writes zero rows due to an empty source query counts as a success. A notebook activity that catches an exception internally and exits cleanly returns success to the pipeline even if no data was written. In Fabric pipelines, activity success status reflects whether the activity completed without a runtime exception, not whether the business outcome was achieved. These are different things.

F

THE FIX

Add explicit outcome validation as a pipeline activity after every critical write operation. A simple Lookup activity that counts rows in the target partition and fails via a Fail activity if the count is zero is more reliable than relying on activity success status alone. Review all activity dependency conditions and change any continue on failure settings that are masking legitimate errors. Treat zero rows written as a failure condition for any pipeline that is expected to load data.

ERROR HANDLING AND MONITORING

35 Your pipeline activity log shows success but the output file is empty or zero bytes.

S

THE SITUATION

A copy activity or pipeline that writes output files to OneLake or an external sink reports success in the monitoring view with a green status. But the target location contains a file with zero bytes or the file is entirely absent. Downstream processes that read from this location fail or produce empty results.

W

WRONG THINKING

Engineers trust the green success status and assume the downstream failure is caused by a read permission issue or a path misconfiguration in the consuming process. The pipeline activity output is not examined in detail because the top-level status shows success. The possibility that a successful activity can write zero bytes is not considered.

H

WHAT IS ACTUALLY HAPPENING

The copy activity completed its execution without a runtime error but the source query or source path returned zero rows or zero bytes. Writing zero bytes to a sink is a valid operation from the pipeline engine's perspective and does not raise an error. The activity succeeded in the sense that it executed without exception. It failed in the business sense that no data was written. These two definitions of success are not the same and the monitoring view only reflects the former.

F

THE FIX

After every copy activity that is expected to write data, add a Get Metadata activity that checks the size or row count of the output. If the size is zero bytes or the row count is zero, route the pipeline to a Fail activity with a descriptive error message rather than allowing downstream activities to consume an empty output. This validation step is the only reliable way to distinguish a successful write from a zero-byte write that the monitoring view treats identically. Add this check as a standard pattern to every pipeline that writes files or tables where empty output is a failure condition.

ERROR HANDLING AND MONITORING

36 A failed activity is retried 3 times. All retries hit the same transient error.

S

THE SITUATION

A copy activity is configured with 3 automatic retries on failure. When a transient network error occurs, the activity retries as expected. But all 3 retries fail with the same error at the same point in execution. The pipeline fails after exhausting retries and the error is treated as a transient issue that should have been resolved by retry.

W

WRONG THINKING

Engineers increase the retry count to 5 assuming more retries will eventually succeed. With 5 retries the pipeline still fails on every attempt. The retry mechanism is blamed as ineffective rather than investigating why the error is consistent across every retry.

H

WHAT IS ACTUALLY HAPPENING

The retry interval is set to zero or near zero seconds. Each retry fires almost immediately after the previous failure. If the underlying cause is a resource limit, a rate limit on the source API, or a capacity constraint that requires time to recover, retrying immediately hits the same condition that caused the original failure. Immediate retries against a resource that needs time to recover will fail consistently regardless of how many retries are configured. The retry count is not the problem. The retry interval is.

F

THE FIX

Set the retry interval to a meaningful value based on the expected recovery time of the failure condition. For API rate limit errors, use an interval that matches the rate limit reset window, typically 60 seconds or more. For capacity-related failures, use intervals of 2 to 5 minutes. For truly transient network errors, 30 seconds is usually sufficient. Use exponential backoff where possible: first retry after 30 seconds, second after 60, third after 120. Also verify the error type before increasing retries. If the same deterministic error appears on every retry, the issue is not transient and retries will never help.

ERROR HANDLING AND MONITORING

37 An activity fails silently. Downstream activities run on wrong data.

S

THE SITUATION

A pipeline with 8 sequential activities completes successfully from end to end. But the final output is wrong. Tracing back through the run history, one activity in the middle produced incorrect output due to a logic error but did not fail. All subsequent activities consumed the wrong data and produced wrong results, all while reporting success.

W

WRONG THINKING

Engineers investigate the final output and trace the error backward through the pipeline. They find the problematic activity eventually but the investigation takes hours because every activity shows success status and there is no signal in the monitoring view pointing to the specific step where data quality diverged.

H

WHAT IS ACTUALLY HAPPENING

Fabric pipeline activities report success or failure based on execution completion, not on data quality. An activity that runs a transformation producing zero matching rows, applies an incorrect filter that drops 80 percent of data, or writes to the wrong partition will report success as long as it completes without a runtime exception. There is no built-in data quality gate between activities. Once wrong data enters the pipeline at any step, every downstream activity consumes it silently.

F

THE FIX

Implement data quality checkpoints between critical pipeline stages. After each major transformation activity, add a validation activity that checks row counts, null rates in key columns, and expected value ranges. Use a Fail activity to stop the pipeline explicitly if any check fails rather than allowing downstream activities to run on bad data. The cost of adding validation activities is small compared to the cost of diagnosing why the final output is wrong after the pipeline has run to completion.

ERROR HANDLING AND MONITORING

38 Your pipeline run history shows succeeded. The target table is empty.

S

THE SITUATION

A pipeline that loads data to a Lakehouse table runs successfully according to the monitoring history. Row counts in the activity output look normal. But querying the target table returns zero rows. The table existed and had data before the pipeline ran.

W

WRONG THINKING

Engineers assume the target table was accidentally dropped or that a permissions issue is preventing reads. When the table is confirmed to exist and reads work correctly, the focus shifts to whether the pipeline wrote to a different table than expected. The possibility that the pipeline overwrote the table with an empty result is not immediately considered.

H

WHAT IS ACTUALLY HAPPENING

The pipeline ran a copy or notebook activity that executed successfully and wrote zero rows. This can happen when the source query returned no results due to an incorrect filter, an empty staging location, or a watermark that moved beyond the available data. The write succeeded because writing zero rows is a valid operation. If the write mode was overwrite, the existing data in the target table was replaced with an empty result. The pipeline succeeded because from an execution standpoint, writing nothing is indistinguishable from writing data.

F

THE FIX

Add a row count check as the first step after every write activity. If rows written is zero, treat it as a failure condition and stop the pipeline before downstream activities consume empty data. For overwrite mode writes specifically, add a pre-write validation that confirms the source contains the expected minimum row count before executing the write. Store the expected row count range in a pipeline parameter and validate against it on every run to catch unexpected source data gaps before they reach the target.

ERROR HANDLING AND MONITORING

39 A long-running pipeline shows no progress for 20 minutes then completes instantly.

S

THE SITUATION

A pipeline that typically takes 25 minutes appears frozen in the monitoring view for 20 minutes with no activity updates. Engineers assume it has hung and are about to cancel it when it suddenly completes and shows success. This pattern repeats on every run.

W

WRONG THINKING

Engineers assume the monitoring UI has a refresh lag and accept the frozen appearance as a cosmetic issue. They do not investigate further because the pipeline completes correctly. The 20-minute blank period is treated as a display problem rather than a signal about execution behavior.

H

WHAT IS ACTUALLY HAPPENING

The pipeline is spending most of its runtime in a Spark session startup phase or in a long-running notebook activity where progress is not being reported back to the pipeline monitoring in real time. Spark notebook activities in Fabric report their status to the pipeline monitoring only at completion, not during execution. A 20-minute notebook run looks identical to a hung pipeline from the pipeline monitoring perspective until the notebook finishes and reports back. The instant completion appearance happens because the status updates in a batch when the notebook closes.

F

THE FIX

Add explicit progress logging inside long-running notebook activities using display output or print statements at meaningful checkpoints. These appear in the notebook run output and can be monitored separately from the pipeline monitoring view. For pipelines where execution time predictability matters, split long notebook activities into multiple shorter activities so each reports its completion status independently to the pipeline. This gives a more accurate picture of pipeline progress without waiting for a single long activity to complete.

ERROR HANDLING AND MONITORING

40 Rerunning a failed pipeline from the failed activity reruns from the beginning.

S

THE SITUATION

A pipeline with 15 activities fails at activity 12 after 35 minutes of successful execution. The Fabric UI offers a rerun option. Engineers select rerun from failed activity expecting the pipeline to resume from activity 12. Instead, the pipeline starts from activity 1 and reruns the entire 35 minutes of already-completed work.

W

WRONG THINKING

Engineers assume the rerun from failed activity feature is broken and raise a support request. The support team confirms the feature is working as designed. The explanation does not fully clarify why a rerun from the failed activity still reruns earlier activities.

H

WHAT IS ACTUALLY HAPPENING

Rerun from failed activity in Fabric pipelines reruns the failed activity and all activities that depend on it in the execution graph. If earlier activities are connected to the failed activity through a dependency chain, they are included in the rerun. Additionally, if the pipeline uses variable values that were set by earlier activities, those variables need to be recomputed. Activities in parallel branches that completed successfully may also be rerun if the execution graph cannot determine their outputs are still valid. The rerun scope is determined by the dependency graph, not simply by the position of the failed activity in a linear sequence.

F

THE FIX

Design pipelines so that long completed work is not in the dependency chain of activities that are likely to fail. Split pipelines into separate parent and child pipelines using the Execute Pipeline activity so that a failure in the child pipeline can be retried independently without rerunning the parent. For pipelines with expensive early activities, write intermediate results to a control table and add a check at the start of subsequent runs that skips already-completed stages based on stored state rather than rerunning them.

06

FABRIC ENGINEERING PATTERNS

Chapter 6 Connections and Configuration

Eight scenarios covering how connection credentials, pipeline parameters, dynamic expressions, and workspace configuration create failures that only appear in production.

ERROR HANDLING AND MONITORING

41 Your pipeline monitoring shows a run in progress that already finished.

S

THE SITUATION

The Fabric Monitoring Hub shows a pipeline run as in progress with a spinning indicator. The run has actually been finished for over an hour. Refreshing the page does not update the status. The run appears stuck in the monitoring view while downstream data is already updated correctly.

W

WRONG THINKING

Engineers assume the pipeline is genuinely still running and wait before investigating downstream. Some attempt to cancel the phantom run, which either fails or creates a confusing state in the monitoring history. Time is wasted waiting for a run that has already completed.

H

WHAT IS ACTUALLY HAPPENING

The pipeline completed successfully but the monitoring status update did not propagate to the Fabric Monitoring Hub before the engineer checked. The Hub has an eventual consistency model where run status updates can lag behind actual execution state by several minutes, especially during periods of high workspace activity. The run completed, wrote data correctly, and closed its session but the monitoring record had not yet refreshed to reflect the final status.

F

THE FIX

Do not rely on the Monitoring Hub as a real-time execution view for time-sensitive operations. Use the pipeline run output and the target table row count as the authoritative completion signal rather than the monitoring status indicator. For automated downstream dependencies, use the Execute Pipeline activity with a wait for completion setting rather than polling the monitoring API for status. Accept that monitoring display lag is a normal characteristic of the platform and design workflows accordingly.

ERROR HANDLING AND MONITORING

42 Your pipeline succeeds in under 1 minute. The data is incomplete.

S

THE SITUATION

A pipeline that typically takes 8 minutes completes in under 60 seconds. No error is raised. The target table has far fewer rows than expected. The speed feels like an improvement but the data tells a different story.

W

WRONG THINKING

Engineers initially think performance improved. When downstream reports flag missing data, the investigation begins. The assumption is a source data issue since the pipeline succeeded quickly with no errors. The connection between the abnormally fast runtime and the data completeness problem is not made immediately.

H

WHAT IS ACTUALLY HAPPENING

The pipeline hit an error condition very early in execution that caused it to exit the main processing branch silently. A common cause is a Lookup activity at the start of the pipeline that returned zero rows, causing an If Condition activity to route to an empty branch that does nothing and completes immediately. Another cause is a Set Variable activity receiving a null value that causes subsequent dynamic expressions to evaluate to empty strings, resulting in copy activities that query with a filter returning no rows. The pipeline completed all its activities successfully but most of them did nothing.

F

THE FIX

Add a dedicated pipeline health check as the very first activity. This check validates that all preconditions are met: the source has data, the watermark value is within expected bounds, and all required configuration values are non-null. If any precondition fails, use a Fail activity to stop the pipeline with a descriptive error rather than allowing it to proceed silently into a do-nothing path. Any pipeline completing significantly faster than its baseline runtime should be treated as a warning signal, not a success.

CONNECTIONS AND CONFIGURATION

43 Your connection works in test. It fails when the pipeline runs on schedule.

S

THE SITUATION

A connection to an external source system is tested successfully in the Fabric connection manager. The pipeline runs correctly when triggered manually. When the same pipeline runs on schedule overnight, the connection fails with an authentication error. Manual runs continue to work the next morning.

W

WRONG THINKING

Engineers test the connection again after the scheduled failure, find it working, and assume a transient network issue caused the overnight failure. They add a retry to the activity and the next scheduled run fails again at the same point. The retry succeeds on the second attempt some nights but not others.

H

WHAT IS ACTUALLY HAPPENING

The connection uses a credential type that requires interactive token refresh, such as OAuth with a delegated user permission flow. When a user manually triggers the pipeline, their active session provides a valid token. When the scheduled trigger fires overnight with no active user session, the token has expired and cannot be refreshed without user interaction. The connection test works because it runs in the context of the logged-in user. The scheduled run fails because no user context is available to refresh the expired token.

F

THE FIX

Replace delegated user credentials with a non-user authentication method for all connections used in scheduled pipelines. Fabric offers two modern options. (1) **Workspace Identity** is the Fabric-native answer: an automatically managed service principal that Fabric creates per workspace, where Microsoft Entra handles credential rotation invisibly. No secret to manage, no expiry to track. Supported for OneLake shortcuts, data pipelines, semantic models, Dataflows Gen2, and increasingly Notebook and Spark job activities. Prefer this when the target data source supports it. (2) **Service principal** credentials for sources or scenarios where Workspace Identity is not supported (typically external authentication targets). Store the secret in Azure Key Vault and rotate on a schedule. Audit all connections in your workspace, identify any that use personal user OAuth tokens, and migrate them before they expire.

CONNECTIONS AND CONFIGURATION

44 Your connection credential expired. Every pipeline using it fails with a misleading error.

S

THE SITUATION

Multiple pipelines start failing simultaneously with errors that appear unrelated: one shows a copy failure, another shows a notebook connection error, a third shows a lookup timeout. The failures all started on the same day. The error messages point to different activities in different pipelines and the common cause is not immediately obvious.

W

WRONG THINKING

Engineers investigate each failing pipeline independently, spending hours on each one before realizing all failures started on the same day. The different error messages make it look like separate unrelated incidents rather than a single root cause affecting all pipelines through a shared dependency.

H

WHAT IS ACTUALLY HAPPENING

All the failing pipelines share a connection to the same data source. That connection uses a credential, typically a service principal client secret or an API key, that expired on the day the failures started. When the credential expires, the connection cannot authenticate and every pipeline using it fails. The error messages vary because different activity types surface the authentication failure differently: a copy activity shows a connection error, a notebook shows an import failure, and a lookup shows a timeout. The root cause is identical but the surface manifestation differs by activity type.

F

THE FIX

Maintain a credential expiry calendar for all service principal secrets, API keys, and certificates used in Fabric connections. Set calendar reminders 30 and 7 days before each expiry. Rotate credentials before they expire rather than after they cause failures. After rotating a credential, test all pipelines that use the affected connection before the next scheduled run. Consider using managed identities where the data source supports them, as managed identities do not have expiry dates and eliminate credential rotation entirely.

CONNECTIONS AND CONFIGURATION

45 Your pipeline connects to an API fine interactively. Scheduled runs get 401 errors.

S

THE SITUATION

A pipeline that calls an external REST API works correctly when run manually during business hours. Scheduled runs outside business hours consistently fail with 401 unauthorized errors. The API credentials have not changed. The same credentials work when the pipeline is run manually immediately after a scheduled failure.

W

WRONG THINKING

Engineers assume the API has rate limits that reset during business hours and that the scheduled runs are hitting those limits. They reduce the scheduled frequency but the 401 errors continue. Rate limiting is ruled out but no other explanation is found and the failures continue.

H

WHAT IS ACTUALLY HAPPENING

The API uses OAuth 2.0 with a token that expires after a fixed duration, typically 1 hour. When a user manually runs the pipeline, a fresh token is obtained using the user credentials at runtime. The scheduled run uses a cached token that was valid when the pipeline was last manually run but has expired by the time the scheduled run executes. A 401 error is returned because the cached token is invalid, not because the credentials are wrong. The token refresh mechanism is not configured for unattended scheduled execution.

F

THE FIX

Configure the connection to use client credentials flow with a service principal rather than an OAuth authorization code flow that depends on user interaction for token refresh. Client credentials flow generates a new token automatically on each request without requiring a user session. If the API only supports user-delegated OAuth, implement a token refresh activity at the start of the pipeline that obtains a fresh token using stored refresh token credentials before the main API calls execute.

CONNECTIONS AND CONFIGURATION

46 You updated a connection. All pipelines using it broke without warning.

S

THE SITUATION

A connection to a source database was updated to point to a new server after a migration. The update was made in the Fabric connection manager and tested successfully. Within hours, multiple pipelines start failing with connection errors. The connection test still shows green in the connection manager.

W

WRONG THINKING

Engineers retest the connection and find it working. They assume the pipelines are caching old connection details and republish each pipeline individually. Some pipelines start working after republish. Others continue to fail. The inconsistency suggests something beyond a simple cache issue.

H

WHAT IS ACTUALLY HAPPENING

Fabric pipelines reference connections by connection ID. When a connection is updated in the connection manager, the change takes effect for the connection object but pipelines that have the connection details embedded in their activity configurations rather than referenced dynamically may retain the old configuration until explicitly republished. Additionally, pipelines running at the moment of the connection update may be mid-execution using the old connection and fail when they attempt to use connection state that changed underneath them.

F

THE FIX

Before updating any shared connection, identify all pipelines that reference it using the Fabric workspace lineage view. Schedule the connection update during a maintenance window when no pipelines are actively running. After updating the connection, republish all affected pipelines and run each one manually to verify the updated connection works end to end before the next scheduled run. For critical shared connections, use connection references as pipeline parameters rather than embedding connection IDs in activity configurations, so updates propagate without requiring individual pipeline republishing.

CONNECTIONS AND CONFIGURATION

47 Your parameterized pipeline works in dev. It fails in prod with the same parameters.

S

THE SITUATION

A pipeline designed with parameters for environment-specific configuration works correctly in the development workspace when parameters are passed at trigger time. The same pipeline deployed to the production workspace with identical parameter values fails immediately. The error references a resource that does not exist.

W

WRONG THINKING

Engineers compare the parameter values between dev and prod runs and confirm they are identical. They assume a deployment issue and redeploy the pipeline. The failure continues. The parameter values look correct to every reviewer who checks them.

H

WHAT IS ACTUALLY HAPPENING

The pipeline has hardcoded references mixed in with parameterized references. Some activity configurations use the pipeline parameter correctly while others reference resource names, table names, or paths that were hardcoded to point at dev resources during development and were never parameterized. In dev these hardcoded references resolve correctly because the dev resource exists. In prod the same hardcoded references point at dev resources that either do not exist in the prod workspace or are not accessible from it.

F

THE FIX

Audit every activity configuration in the pipeline for hardcoded strings that should be parameters. Use the pipeline parameter panel to find all parameter references and then manually check each activity for any remaining literal strings that reference environment-specific resources. A reliable approach is to search for dev workspace names, dev Lakehouse names, and dev server addresses in the pipeline JSON definition before promoting to production. For systematic parameterization across pipelines, use Fabric **Variable Libraries**: a Variable Library defines named variables (connection strings, workspace IDs, source database names) with value sets per environment, and pipelines reference the variables by name. When a pipeline is promoted, the variable resolves to the target environment's value automatically. This replaces ad hoc per-pipeline parameter discipline with a workspace-level configuration surface. Establish a convention that no environment-specific string may appear as a literal in any activity configuration.

CONNECTIONS AND CONFIGURATION

48 A pipeline parameter default value overrides what you pass at runtime.

S

THE SITUATION

A pipeline accepts a date parameter used to control the incremental load window. When the pipeline is triggered manually with a specific date value, the output consistently reflects a different date. The date in the output matches the parameter default value defined in the pipeline, not the value passed at trigger time.

W

WRONG THINKING

Engineers confirm the correct date is being passed in the trigger payload by checking the trigger configuration. They assume a pipeline expression bug is overwriting the parameter value partway through execution. The investigation focuses on the transformation logic rather than the parameter binding itself.

H

WHAT IS ACTUALLY HAPPENING

The pipeline activities reference a variable rather than the parameter directly. A Set Variable activity at the start of the pipeline initializes the variable using an expression that falls back to the parameter default value rather than the passed value, either due to an incorrect expression syntax or because the variable is initialized before the parameter binding is evaluated. In Fabric pipelines, variables and parameters are separate concepts. If an activity sets a variable using a hardcoded default instead of referencing the parameter correctly, the runtime-passed parameter value is never used.

F

THE FIX

Always reference parameters directly in activity configurations using the correct expression syntax. Verify the binding by adding a Set Variable activity that captures the parameter value and then outputting it via a Web activity or a notebook print statement in dev runs to confirm the correct value is being received. Review any Set Variable activities that initialize variables from parameters to confirm they use the parameter reference expression and not a fallback literal. Test pipeline behavior by passing parameter values that differ clearly from the default, making it immediately obvious if the default is being used instead.

CONNECTIONS AND CONFIGURATION

49 Your dynamic content expression evaluates correctly in test. Wrong value in run.

S

THE SITUATION

A dynamic content expression used in a copy activity source query or a file path is validated in the expression builder and shows the correct expected value. When the pipeline runs, the activity uses a different value than the expression preview showed. The discrepancy causes the copy to read from the wrong source or write to the wrong path.

W

WRONG THINKING

Engineers recheck the expression in the builder after the failed run and confirm it still evaluates to the correct value. They assume a platform caching issue and rerun the pipeline. The wrong value persists across multiple runs. The expression appears correct in every manual evaluation.

H

WHAT IS ACTUALLY HAPPENING

The expression uses a function like `utcNow()` or `formatDateTime()` that returns a value based on the exact moment of evaluation. In the expression builder, the preview evaluates at the moment you click validate. In the actual pipeline run, the function evaluates at activity execution time, which may be seconds or minutes later depending on pipeline startup time and preceding activity duration. If the expression generates a date-based path like a year-month-day folder, and the pipeline runs across midnight, the expression in earlier activities may produce today while later activities produce tomorrow.

F

THE FIX

For any expression that generates time-based values used across multiple activities, evaluate the expression once at the start of the pipeline using a Set Variable activity and store the result in a pipeline variable. Reference the variable in all subsequent activities rather than re-evaluating the expression independently in each activity. This ensures all activities use the same consistent value regardless of when they execute within the pipeline run.

CONNECTIONS AND CONFIGURATION

50 You copy a pipeline across workspaces. Connections point to the wrong environment.

S

THE SITUATION

A pipeline is exported from the development workspace and imported into the production workspace. The pipeline appears correctly in the production workspace. When run, it reads from development data sources and writes to development targets despite being in the production workspace. Production data is untouched.

W

WRONG THINKING

Engineers check the pipeline activities and the visible configuration looks correct. They assume the import process failed silently and reimport the pipeline. The same behavior repeats. The pipeline in production is genuinely executing against development resources without any visible indication in the standard pipeline view.

H

WHAT IS ACTUALLY HAPPENING

When a pipeline is exported and imported across workspaces, connection references embedded in activity configurations carry over from the source workspace. These connection IDs resolve to the development connections in the production workspace if connection IDs were not remapped during import. The pipeline runs successfully because the development connections are valid and accessible, but all data movement goes to and from the development environment. There is no automatic environment detection or connection remapping during import unless explicitly configured.

F

THE FIX

Before running any imported pipeline in production, open each activity and verify that all connection references point to production connections. Use the Fabric workspace lineage or the pipeline JSON to audit every connection ID referenced in the pipeline against the list of production connections. The Fabric-native pattern for this is two-layered: use **Fabric deployment pipelines** to promote artifacts across workspace stages (dev → test → prod), and use **Variable Libraries** to hold the environment-specific values (connection strings, workspace IDs) that need to differ per stage. Note that Fabric deployment pipelines support data source rules and parameter rules for some artifact types (semantic models, dataflows Gen1, notebooks, paginated reports) but not for data pipelines or warehouses — for those, Variable Libraries are the supported parameterization mechanism. Manual remapping after import is the fragile fallback when the artifact type is not supported by either; treat it as a known weak spot in your deployment process and document it explicitly.

CONCLUSION

Pipelines fail at the boundaries.

Across 50 scenarios, one pattern repeats. Pipeline failures rarely happen in the middle of a well-tested transformation. They happen at the boundaries: between the trigger and the run, between the source and the copy, between the identity used in development and the identity used in production. Building reliable pipelines in Fabric means treating validation as a first-class activity, not an afterthought. Success status is not enough. A pipeline that reports success can still produce wrong data, miss rows, or process the wrong time window.

ALSO IN THE SERIES

Warehouse and SQL

> Your SCD Type 2 MERGE is creating duplicate current records.
> Your query ran in 2 seconds last week. 3 minutes this week. Nothing changed.

Power BI in Fabric

> Your Direct Lake report fell back to DirectQuery silently after a schema change.
> RLS is defined correctly. Users are still seeing data they should not.

And more. Free at **ssanjaychandra.com**

A NOTE ON THIS RELEASE

This is the initial release of the Fabric Engineering Patterns series. The patterns and fixes reflect the platform as it stands today. Microsoft Fabric evolves quickly and some guidance may need updating as the platform matures. If you spot a technical inaccuracy, a pattern that has been superseded by a platform update, or a scenario that deserves a deeper treatment, your feedback is genuinely welcome. Reach out through any of the channels below.

ACKNOWLEDGEMENTS

Claude (Anthropic) was used as a writing assistant while putting this book together.

CONNECT

🌐 www.ssanjaychandra.com
💼 linkedin.com/in/ssanjaychandra
✉️ ssanjaychandra@outlook.com

Download all books in this series free at ssanjaychandra.com/freedownloads