FABRIC ENGINEERING PATTERNS

Lakehouse and PySpark

Common Patterns. Clear Fixes.

50 engineering patterns covering Delta Lake internals, Spark performance, Lakehouse design, and session concurrency in Microsoft Fabric. Patterns every Fabric engineer must recognise, covering the gap between how Fabric is described and how it actually behaves under real-world load.

Download all books free at ssanjaychandra.com/freedownloads

AUTHOR
Sanjay Chandra

50
SCENARIOS

INTRODUCTION

Why production breaks differently.

Most Fabric learning resources teach you how to build things. This book teaches you what breaks after you build them.

Every scenario in this book came from a real pattern that engineers encounter in production. The wrong answer in each case is not a naive guess. It is the answer that feels correct based on a reasonable understanding of the technology. That is what makes these scenarios worth studying.

The book is organised into four chapters. Chapter 1 covers Delta Lake internals, specifically the behaviours that are invisible during development but become expensive at scale. Chapter 2 covers Spark performance, focusing on the patterns that explain why a job that worked perfectly in dev fails or slows down dramatically in production. Chapter 3 covers Lakehouse design decisions that compound over time. Chapter 4 covers session and concurrency issues that only appear when multiple workloads share the same Fabric capacity.

Each scenario follows the same structure: the situation, the wrong thinking, what is actually happening, and the fix. The wrong thinking section is as important as the fix. Understanding why a wrong answer is tempting is the difference between fixing a specific problem and developing the intuition to avoid the next one.

Microsoft Fabric is evolving quickly. The scenarios in this book reflect production patterns observed on Fabric as of 2024 and 2025. Where Fabric-specific behaviour differs from general Spark or Delta Lake behaviour, that distinction is called out explicitly.

Read it cover to cover or use it as a reference when something breaks. Either way, the goal is the same: to give you the mental model you need to diagnose engineering problems faster and build more resilient Fabric solutions from the start.

Reflects the state of Microsoft Fabric as of Q2 2026. Where Fabric-specific behaviour differs from general Spark or Delta Lake behaviour, that distinction is called out explicitly. Behaviours marked as current may change as the platform evolves.

DIAGNOSTIC INDEX

Start with the symptom. Not the chapter.

When a Spark job or Delta table misbehaves 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.

Table fast last month, slow now (no code change).
Check 01 (small-file accumulation), 22 (compounding maintenance failure), 10 (stale stats), 03 (no Z-order).

Job hangs or OOMs on large data, fine on small.
Check 11 (aggregation skew), 16 (1M→100M scaling), 18 (OOM via shuffle), 34 (scaling failure modes).

DELETE or MERGE on large table is rewriting everything.
Check 02 (DELETE storage), 05 (MERGE rewrites), 39 (partial-column writes). Copy-on-write semantics.

Join shuffling more than expected.
Check 10 (stale stats), 13 (skew). Refresh stats with `ANALYZE TABLE`; broadcast hint if one side is small.

Concurrent writes failing or producing missing data.
Check 21 (full overwrites), 33 (commit conflicts), 45 (read-after-write).

Shortcut performance worse than native table.
Check 23. Source-type matters: ADLS shortcuts lack caching; S3/GCS/gateway shortcuts have it.

First run of the day is slow, then fast.
Check 48 (Spark cold start). Warm-up notebook pattern; High Concurrency Mode for notebook-heavy pipelines.

One task takes 10x longer than the others.
Check 13 (data skew), 17 (apparent vs real parallelism). Same root cause — different angles.

Spark UI shows many shuffle partitions and small tasks.
Check 41 (too many partitions = overhead). Target 128–256MB per shuffle partition.

Time travel query suddenly fails.
Check 07 (VACUUM retention shorter than time-travel need). Once files are gone, no recovery.

Pipeline succeeded but the data is wrong.
Check 47 (retry duplicates), 06 (schema drift), 30 (silent partial completion).

Spark job fast alone, slow when others run.
Check 31, 32, 27. Workspace-level capacity contention.

Some scenarios appear under multiple symptoms because one root cause produces several. The principles repeat: maintenance is not optional, skew is the most common Spark performance problem, copy-on-write means small writes can cause large rewrites, and stats become stale.

01

FABRIC ENGINEERING PATTERNS

Chapter 1 Delta Lake Internals

Twelve scenarios covering how Delta tables behave in production and why they stop behaving the way you expect.

01

DELTA LAKE

Your Delta table was fast last month. Now it is slow. Nobody changed the notebook.

THE SITUATION

Same notebook. Same data volume. Same Spark session config. Queries that completed in 40 seconds last month now take 6 minutes. No code was changed, no schema was altered, no new data source was added.

WRONG THINKING

Most engineers assume a Spark session issue and start tweaking configs, raising capacity requests, or blaming the environment. That is the wrong place to look first. The problem is almost never the session.

WHAT IS ACTUALLY HAPPENING

Delta tables accumulate small files silently over time. Every append operation creates new Parquet files in OneLake. After weeks of daily writes, a table that started with 10 files may now have thousands. Spark spends more time on file metadata lookups and planning than on actual computation. The notebook did not get slower. The table did.

THE FIX

Schedule `OPTIMIZE` on the table after every significant write cycle. For large tables, add Z-ordering on your most frequently filtered column. This co-locates related data and reduces scan width dramatically. Follow every `OPTIMIZE` run with `VACUUM` to remove the stale files left behind. Set the retention window explicitly. None of this runs automatically in Fabric. If it is not scheduled, it is not happening.

02

DELTA LAKE

You deleted 50% of your Delta table. Storage in OneLake did not reduce.

THE SITUATION

You ran a delete operation removing half the rows from a large Delta table. You check OneLake storage. The numbers barely moved. Half the data is gone logically but the storage cost looks the same as before.

WRONG THINKING

Many engineers assume delete operations in Delta Lake physically remove data immediately, the same way a traditional database would. They file a support ticket or assume something went wrong with the delete operation itself.

WHAT IS ACTUALLY HAPPENING

Delta Lake uses copy-on-write. When you delete rows, Delta rewrites the affected Parquet files with those rows removed and marks the old files as logically deleted in the transaction log. The old files remain physically in OneLake until you explicitly remove them. This is intentional. It is what makes time travel possible. But it also means storage does not shrink until you act on it.

THE FIX

Run `VACUUM` with an appropriate retention window to physically delete the old files from OneLake. The default retention is 7 days to protect time travel queries. If you do not need 7 days of history on this table, lower it explicitly. For tables with aggressive delete patterns, pair `VACUUM` with `OPTIMIZE` to compact the rewritten files before vacuuming. Storage will not manage itself in Fabric.

03

DELTA LAKE

You ran OPTIMIZE on your Delta table. Queries are still slow.

THE SITUATION

Queries on a Delta table were slow. You ran `OPTIMIZE` expecting a significant improvement. The table compacted. File count dropped. But query times barely changed. You run it again. Same result.

WRONG THINKING

The common assumption is that `OPTIMIZE` alone solves read performance. Engineers run it and expect queries to return to baseline. When they do not, the assumption shifts to Fabric limitations or infrastructure issues.

WHAT IS ACTUALLY HAPPENING

`OPTIMIZE` compacts small files into larger ones but it does not control which data lands in which file unless you tell it to. If your queries filter on a specific column, Spark still has to scan every compacted file unless the data within those files is ordered by that column. Without Z-ordering, `OPTIMIZE` gives you fewer files but not smarter files. File count was not the bottleneck. Data layout was.

THE FIX

Run `OPTIMIZE tablename ZORDER BY (your_filter_column)`. Z-ordering co-locates rows with similar values into the same files, which allows Spark to skip entire files during a filtered scan. Choose the column you filter on most in your queries. Z-ordering on a high-cardinality column like a timestamp works well. Z-ordering on a boolean or low-cardinality column provides little benefit. One well-chosen column is better than many poorly chosen ones.

04

DELTA LAKE

Your Delta table has 500 columns. Read performance is crawling even on simple queries.

THE SITUATION

A wide Delta table with several hundred columns is showing poor read performance even on queries that only touch 5 or 10 of those columns. Row count is not especially large but every query is slow regardless of what columns are selected.

WRONG THINKING

Engineers look at row count and file count first. When those look reasonable, the assumption shifts to Spark session configuration or OneLake network latency. The width of the table is rarely the first suspect.

WHAT IS ACTUALLY HAPPENING

Parquet is columnar, meaning each column is stored separately. When Spark reads a wide Delta table, it still has to process footer metadata for every column in every file even if your query only needs 5 of them. With 500 columns that metadata overhead is significant. Wide tables also often have poor column statistics coverage in the Delta log, which limits Spark's ability to skip files using predicate pushdown.

THE FIX

Always select only the columns you need. Never use `SELECT *` on wide tables in notebooks. If certain column groups are always queried together, consider splitting the table vertically into narrower domain tables joined on a key. Review column statistics using `DESCRIBE DETAIL` and ensure statistics collection is not disabled. For very wide tables with mixed access patterns, evaluate whether pre-projected views fit your workload better than querying the raw table directly.

05

DELTA LAKE

You merged a small dataset into a large Delta table. It rewrote the entire table.

THE SITUATION

You run a daily `MERGE` operation, a small batch of updates against a large Delta table. The merge takes far longer than expected and Spark UI shows a massive write volume. It behaves like a full rewrite every single run.

WRONG THINKING

The common assumption is that `MERGE` is smart enough to only touch files containing matching rows. Engineers expect a small source dataset to produce a proportionally small write. When it does not, they assume it is a Fabric limitation or a bug.

WHAT IS ACTUALLY HAPPENING

Delta Lake's `MERGE` uses copy-on-write. When it finds a file containing at least one matching row, it rewrites the entire file, not just the matching rows. If your large target table is not partitioned on the merge key, Spark scans every file looking for matches and rewrites every file that has even one match. A merge on an unpartitioned 100GB table can rewrite 80GB even if only 1,000 rows changed.

THE FIX

Partition your target Delta table on the column used in your `MERGE` condition, typically a date or region column. Add a partition filter explicitly in your merge condition so Spark prunes partitions before scanning: `target.date = source.date`. This tells Spark to only open files in the relevant partition. For high-frequency merges on large tables, also evaluate whether Liquid Clustering on the merge key column gives better file pruning than static partitioning for your specific access pattern.

06

DELTA LAKE

Schema evolution broke your downstream notebook silently.

THE SITUATION

A source team added three new columns to a Delta table. No alerts fired. No pipelines failed. But two weeks later a downstream notebook is producing wrong aggregations and nobody can trace why. The data looks fine on the surface.

WRONG THINKING

Teams assume schema changes will either break a pipeline loudly or have no effect at all. Silent corruption (where jobs succeed but produce wrong results) is rarely the first hypothesis when something looks off downstream.

WHAT IS ACTUALLY HAPPENING

Delta Lake supports schema evolution but it does not validate whether downstream consumers are ready for that change. When new columns are added and a downstream notebook selects by position rather than by name, or when a column is renamed and the old reference silently returns null instead of failing, the job completes successfully with corrupted output. Delta's mergeSchema option makes ingestion easy but it gives no protection to anything downstream of the write.

THE FIX

Always select columns by name explicitly, never by position or `SELECT *` in notebooks. Add schema validation checks at the start of downstream notebooks using `DESCRIBE TABLE` output compared against an expected schema stored separately. For critical tables, disable automatic schema evolution and require explicit schema migration approvals before changes reach production. Treat schema changes as breaking changes until proven otherwise.

07

DELTA LAKE

Your time travel query worked yesterday. Today it fails.

THE SITUATION

You use Delta time travel regularly to query historical snapshots for auditing and debugging. A query that worked fine yesterday using `VERSION AS OF` or `TIMESTAMP AS OF` now throws an error. Nothing changed in your notebook.

WRONG THINKING

Most engineers assume the error is a syntax issue or a Fabric platform problem. They re-run the query multiple times or check for Fabric service incidents. The real cause is closer to home and entirely preventable.

WHAT IS ACTUALLY HAPPENING

Someone ran `VACUUM` on the table with a retention window shorter than the version you are trying to query. `VACUUM` physically deletes the Parquet files that older Delta versions point to. Once those files are gone, time travel to that version is permanently broken. There is no recovery. The Delta log still has the entry for that version but the underlying files it references no longer exist in OneLake.

THE FIX

Set your `VACUUM` retention window to be longer than your longest time travel requirement. If you need 30 days of history, set retention to at least 30 days. Never set retention below 7 days. Delta Lake itself warns against this. Before running `VACUUM` on any table used for auditing, check who depends on historical versions and what the furthest-back query requirement is. Document the retention policy per table and enforce it through scheduled maintenance notebooks, not ad hoc runs.

08

DELTA LAKE

You have two Delta tables. Same size. One compacts in minutes, one takes forever.

THE SITUATION

Two Delta tables of similar row count and storage size. Both have accumulated small files. You run `OPTIMIZE` on both. One finishes in 4 minutes. The other runs for over an hour and you eventually cancel it.

WRONG THINKING

The assumption is that `OPTIMIZE` runtime scales with table size. If two tables are the same size, they should compact in roughly the same time. When they do not, engineers blame session configuration or Fabric capacity throttling.

WHAT IS ACTUALLY HAPPENING

The slow table has a partition column with extremely high cardinality, for example a UUID or a millisecond timestamp. Delta's `OPTIMIZE` compacts files within each partition independently. If a table has millions of partitions because the partition key is too granular, `OPTIMIZE` has to open, read, and rewrite files across millions of individual partition directories. The work is not proportional to data size. It is proportional to partition count. High cardinality partitioning is one of the most common and expensive Delta mistakes in production.

THE FIX

Partition Delta tables on low to medium cardinality columns only: date, region, product category. Never partition on IDs, UUIDs, or high-precision timestamps. If you need fast lookups on high-cardinality columns, use Z-ordering instead of partitioning. For tables already suffering from over-partitioning, the cleanest fix is a full rewrite using `REPARTITION` on a better column followed by `OPTIMIZE`. Prevention is far cheaper than remediation at scale.

02

FABRIC ENGINEERING PATTERNS

Chapter 2 Spark Performance

Fourteen scenarios covering how Spark behaves under real real-world load and why the Spark UI tells a very different story from what you expect.

09

SPARK PERFORMANCE

Your Spark notebook runs fine in dev. In production it is 10x slower.

THE SITUATION

A notebook tested extensively in dev with clean results and acceptable runtimes. Promoted to production it is consistently 10 times slower. Same code. Same logic. The data volume in production is larger but not dramatically so.

WRONG THINKING

The instinct is to scale up the Spark session or request more Fabric capacity. Throwing compute at the problem feels like the fastest fix. It is almost never the right one and it is always expensive.

WHAT IS ACTUALLY HAPPENING

Dev data is almost always a small clean sample. Production data exposes every assumption you got away with in dev. Joins without partition pruning that now scan 200GB instead of 200MB, shuffle partitions defaulting to 200 that worked fine on small data but create massive skewed tasks on real volumes, and Delta tables in production that have never been optimized unlike the dev tables you manually curated. The notebook is the same. The environment is completely different.

THE FIX

Open the Spark UI and check the DAG before changing anything else. Look at shuffle read and write sizes per stage. Identify which stage is taking the most time. Check task distribution. If one task takes 10x longer than others, you have skew. Verify shuffle partition count with `spark.conf.get("spark.sql.shuffle.partitions")` and tune it based on actual data size. Fix the root cause in the execution plan first. Compute is always the last resort.

10

SPARK PERFORMANCE

Same code, same data. Your join is shuffling 10x more data than last week.

THE SITUATION

A join that ran efficiently last week is now generating enormous shuffle read and write volumes in the Spark UI. Nothing changed in the notebook. The tables being joined are the same size. The query plan looks identical.

WRONG THINKING

Engineers check the notebook for accidental changes, review table sizes, and look at the join condition. When everything looks the same they assume it is a Fabric platform issue or a transient Spark scheduling problem.

WHAT IS ACTUALLY HAPPENING

Column statistics in the Delta log have gone stale. Spark relies on these statistics to choose between a sort-merge join and a broadcast join. When statistics are fresh and accurate, Spark may correctly identify one table as small enough to broadcast. When statistics are stale or missing (which happens after large writes, schema changes, or simply over time), Spark defaults to a sort-merge join, which involves a full shuffle of both tables. AQE can help but only if the data distribution is reasonably predictable. Stale statistics produce bad plans that AQE cannot fully recover from.

THE FIX

Run `ANALYZE TABLE tablename COMPUTE STATISTICS FOR ALL COLUMNS` on both tables involved in the join. This refreshes the Delta log statistics so Spark can make accurate join strategy decisions. Add this to your maintenance schedule alongside `OPTIMIZE` and `VACUUM`. If one table is consistently small enough to broadcast, you can also hint Spark explicitly using `broadcast()` in your PySpark join to override the planner decision regardless of statistics.

11

SPARK PERFORMANCE

Aggregation is fast on small data. On large data it hangs with no error.

THE SITUATION

A groupBy aggregation runs in seconds on dev data. On the full data at scale it runs for hours without completing and without throwing an error. The session stays alive. Progress bars show tasks completing but the job never finishes.

WRONG THINKING

Engineers wait for the job to finish assuming it is just slow. After an hour they cancel and try increasing session memory or reducing the dataset. Neither addresses the real problem which is in the data distribution, not the compute.

WHAT IS ACTUALLY HAPPENING

The aggregation has a severely skewed key. One or a few values in the group-by column appear millions of times while most values appear only a handful of times. Spark assigns one task per partition after the shuffle, so the task handling the dominant key is processing 100x more data than every other task. It runs for hours while all other tasks complete in seconds. The job appears stuck because it is waiting on that one task. AQE's skew join handling addresses join skew specifically but does not always resolve aggregation skew.

THE FIX

First confirm skew by checking the Spark UI tasks tab. Look for one task with dramatically higher shuffle read size than others. Then apply salting: add a random integer suffix to the skewed key before aggregating, perform a partial aggregation, then aggregate again after removing the salt. This distributes the dominant key across multiple partitions. Alternatively, filter out or handle the dominant keys separately and union the results. Increasing shuffle partitions via `spark.sql.shuffle.partitions` can also help spread the load.

12

SPARK PERFORMANCE

Your filter is in the notebook but Spark is still scanning the full table.

THE SITUATION

You have a filter on a date column that should limit the scan to the last 30 days of a multi-year Delta table. The Spark UI shows the job reading the entire table. Query time is the same whether the filter is there or not.

WRONG THINKING

Engineers assume that any filter written in PySpark automatically gets pushed down to the file scan level and prunes unnecessary partitions. When scan volume does not reduce, they assume the filter is not being applied at all and add explicit repartitioning or caching before the filter.

WHAT IS ACTUALLY HAPPENING

Predicate pushdown and partition pruning only work when the filter column matches the Delta table's partition column exactly. If the table is not partitioned on the date column, or if the filter is applied after a transformation that breaks the lineage Spark needs for pushdown, Spark has no choice but to scan every file and apply the filter after reading. A filter in PySpark is a logical operation. Partition pruning is a physical optimization. They are not the same thing.

THE FIX

Verify partition pruning is working by checking the Spark UI files read count. If it matches the total file count, pruning is not happening. Ensure the filter column matches the partition column name exactly, including case. Avoid wrapping partition columns in functions like `year(date_col)` inside filters. This breaks pushdown. If the table is not partitioned on the column you filter most, Z-ordering on that column is the next best option. For critical read paths, use `EXPLAIN` to verify the physical plan shows partition filters.

13

SPARK PERFORMANCE

The job finishes but one task took 10x longer than all the others.

THE SITUATION

A Spark job completes successfully but runtime is far longer than expected. Opening the Spark UI tasks view reveals 199 tasks completing in under 10 seconds each and one task that ran for 18 minutes. The job waited for that one task to finish.

WRONG THINKING

Since the job succeeded, engineers often move on without investigating. When they do look, they assume the slow task hit a bad executor or a transient infrastructure issue and re-run hoping it resolves itself. It rarely does.

WHAT IS ACTUALLY HAPPENING

This is data skew. One partition after the shuffle contains a disproportionately large amount of data compared to all others. The task processing that partition has to do the work of 100 normal tasks on its own. This is almost always caused by a dominant join key or a group-by key with a heavily imbalanced distribution. AQE in Fabric's Spark can detect and split skewed partitions automatically in some cases, but it depends on accurate statistics and the skew threshold configuration being triggered.

THE FIX

In the Spark UI, check the shuffle read size of the slow task versus the median task. A 10x or greater difference confirms skew. Check which join key or group-by key is causing it, often a null value or a single dominant ID. Handle nulls explicitly before the join. For join skew, use `.hint("skew")` on the skewed DataFrame or increase `spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes` to help AQE detect and split the skewed partition. For aggregation skew, apply salting as described in Scenario 11.

14

SPARK PERFORMANCE

Your window function worked fine in dev. It kills the session in production.

THE SITUATION

A window function using `PARTITION BY` and `ORDER BY` runs cleanly in dev. In production the session stalls, memory warnings appear in the logs, and the job eventually fails or takes an unacceptably long time to complete.

WRONG THINKING

Engineers assume window functions are handled efficiently by Spark in the same way aggregations are. Since the same logic worked in dev, the production failure is attributed to session memory limits or Fabric capacity rather than the window function design itself.

WHAT IS ACTUALLY HAPPENING

Window functions require Spark to sort and hold entire partitions in memory before computing the result. If the `PARTITION BY` column has low cardinality in production, for example partitioning by a column that has only 3 or 4 distinct values. Each partition contains an enormous amount of data. Spark has to sort hundreds of millions of rows per partition in memory before it can emit a single result row. In dev with sampled data this is fine. In production it exhausts available memory per executor entirely.

THE FIX

Check the cardinality of your `PARTITION BY` column in data at scale before deploying. If cardinality is low, add a secondary column to the partition to increase granularity and distribute the data more evenly across partitions. Where possible, replace window functions with grouped aggregations joined back to the original DataFrame. This avoids the full sort and is often significantly faster on large datasets. Verify the approach handles your use case before switching, as window semantics and aggregation semantics differ for some operations.

15

SPARK PERFORMANCE

Your job ran in 5 minutes yesterday. Same job, same data, 45 minutes today.

THE SITUATION

A scheduled notebook that runs daily has been consistently completing in 4 to 6 minutes for weeks. Today without any code or data changes it took 45 minutes. It completed successfully. Tomorrow it is back to 5 minutes.

WRONG THINKING

Because the job succeeded and the slowdown appears intermittent, engineers often dismiss it as a transient platform issue. It gets logged, noted, and ignored until it happens again, at which point the investigation starts from scratch each time.

WHAT IS ACTUALLY HAPPENING

In Fabric's serverless Spark model, session startup and resource allocation time varies based on current capacity utilization across the workspace. When other heavy workloads are running simultaneously, other notebooks, pipelines, or reports refreshing. Your job competes for capacity units. The 45-minute run likely spent most of its time waiting for resources to become available before execution even started. The actual computation time was still 5 minutes. This is a concurrency and scheduling issue, not a code or data issue.

THE FIX

Check the Spark UI job start time versus submission time. A large gap confirms the job waited for resource allocation. Schedule resource-intensive notebooks during off-peak hours when workspace capacity utilization is lower. For time-sensitive jobs, review what else is scheduled at the same time and stagger execution. In Fabric, capacity is shared across all workloads in the workspace. Understanding the concurrency profile of your workspace is as important as optimizing individual notebook performance.

16

SPARK PERFORMANCE

Your job runs fine at 1 million rows. It falls over at 100 million.

THE SITUATION

A notebook tested and validated on a 1 million row sample runs cleanly in every test. When pointed at the full 100 million row data at scale it either fails outright or runs so slowly it becomes unusable. The logic is identical.

WRONG THINKING

The assumption is that a 100x increase in data should produce roughly a 100x increase in runtime. Engineers expect linear scaling and are surprised when the job does not just run slower but collapses entirely at scale.

WHAT IS ACTUALLY HAPPENING

Several Spark behaviors that are invisible at small scale become critical at large scale. Shuffle partitions set to 200 work fine at 1 million rows but create 200 partitions each containing 500,000 rows at 100 million rows, overwhelming individual tasks. Joins that Spark chose to broadcast at small scale now exceed the broadcast threshold and fall back to expensive sort-merge joins with full shuffles. Data skew that was statistically insignificant in the sample becomes catastrophic at full volume. None of these problems are visible until real data is used.

THE FIX

Never validate notebooks on sampled data only. Test on at least 20 to 30 percent of production volume before promoting. Tune `spark.sql.shuffle.partitions` based on actual data size, targeting roughly 128MB to 256MB per partition. Check the Spark UI physical plan on full data to verify join strategies match expectations. Add explicit data distribution checks at the start of notebooks to catch skew before it reaches a join or aggregation stage.

17

SPARK PERFORMANCE

Your Spark job has 1000 tasks. Only a fraction are doing real work.

THE SITUATION

The Spark UI shows 1000 tasks created for a job. But looking at the timeline, most tasks complete near instantly while a small number run for minutes. Total CPU utilization across the session looks low relative to the task count.

WRONG THINKING

More tasks typically feels like more parallelism. Engineers assume a high task count means the job is well distributed. A low overall runtime would confirm this. But task count and useful parallelism are not the same thing.

WHAT IS ACTUALLY HAPPENING

The data is heavily skewed across partitions. Most of the 1000 partitions contain very few rows and complete their tasks almost immediately. A handful of partitions contain the majority of the data and their tasks run for the entire job duration. The job is effectively sequential for the heavy partitions despite appearing highly parallel. This is the same skew problem manifesting differently. High task count with short median task duration and long tail task duration is the signature pattern in the Spark UI.

THE FIX

In the Spark UI, look at the task duration distribution on the stage summary. A large gap between median and maximum duration confirms skew. Use `repartition()` on a well-distributed column before the heavy stage to redistribute data more evenly. If the skew is in a join, enable AQE skew join handling and verify the skew threshold is set appropriately for your data size. The goal is not more tasks but more evenly sized tasks.

18

SPARK PERFORMANCE

Your job fails with OOM. There is no obvious memory issue in the notebook.

THE SITUATION

A notebook fails mid-execution with an out of memory error. The notebook looks clean. There are no obvious large collections being built, no loops accumulating data, and no explicit caching. The error appears inconsistently, sometimes on stage 3, sometimes on stage 7.

WRONG THINKING

Engineers look for the obvious culprits first: large variables, explicit caches, or data being pulled to the driver. When none are found they assume Fabric's serverless session has insufficient memory for the workload and request a larger session configuration.

WHAT IS ACTUALLY HAPPENING

Spark is spilling shuffle data to disk and then running out of both memory and disk space during a large shuffle operation. In Fabric's serverless model, each session has defined memory limits per executor slot. When shuffle partitions are too few, individual partitions become very large and the memory required to sort and process them exceeds what is available per executor. The inconsistency in which stage fails depends on which stage produces the largest shuffle output on that particular run.

THE FIX

Increase `spark.sql.shuffle.partitions` to create more, smaller partitions that fit comfortably in memory per executor slot. A practical starting point is to target 128MB to 256MB per shuffle partition. Check the Spark UI for spill metrics on each stage. If spill to disk is occurring before the OOM, that confirms the partition size is the issue. Also review whether any intermediate DataFrames are being persisted unnecessarily with `cache()` or `persist()` and unpersist them as soon as they are no longer needed.

03

FABRIC ENGINEERING PATTERNS

Chapter 3 Lakehouse Design

Thirteen scenarios covering how Lakehouse architecture decisions made early create performance and reliability problems that compound over time.

19

LAKEHOUSE DESIGN

Two notebooks read the same Delta table. One is fast, one is consistently slow.

THE SITUATION

Two notebooks both read from the same Delta table in the same Lakehouse. One consistently returns results in under a minute. The other takes 10 to 15 minutes for what appears to be a similar operation. Both are reading the same data source.

WRONG THINKING

Since the table is identical for both, engineers assume the difference must be in session configuration or scheduling. They compare session settings, find minor differences, and spend time tuning memory and core counts without addressing the real cause.

WHAT IS ACTUALLY HAPPENING

The fast notebook reads only the columns it needs with a precise filter that aligns with the table partition column, allowing Spark to prune most files before reading. The slow notebook uses a broad select with filters on non-partition columns, forcing a full table scan every time. The difference is not session configuration. It is query design. Two notebooks reading the same table can have radically different physical execution plans depending on how the read is constructed.

THE FIX

Audit both notebooks using `EXPLAIN` or the Spark UI physical plan view. Confirm partition pruning is happening in the fast notebook and not in the slow one. Rewrite the slow notebook to select only required columns and to filter on the partition column first. If the slow notebook genuinely needs a full scan, consider whether a pre-aggregated or pre-filtered Delta table maintained separately would serve that use case better than reading the raw table every time.

20

LAKEHOUSE DESIGN

You added a partition column to your Delta table. Queries on other columns got slower.

THE SITUATION

You partitioned a large Delta table on a date column expecting across-the-board performance improvements. Queries filtering by date are now faster. But queries filtering by other columns like customer ID or product category are noticeably slower than before partitioning.

WRONG THINKING

Partitioning is understood as a universal performance improvement. Once a table is partitioned, engineers expect all queries to benefit regardless of which columns are in the filter. The idea that partitioning can hurt some queries is counterintuitive.

WHAT IS ACTUALLY HAPPENING

Partitioning physically separates data into directories by the partition column value. When you query on a different column, Spark has no partition pruning available and must scan every partition directory. Before partitioning, the data was in fewer larger files and file-level statistics were more effective. After partitioning, the same data is spread across hundreds of directories, each with many small files, and Spark must open metadata for all of them. Partitioning optimizes for one access pattern at the expense of all others.

THE FIX

Partition only on the single column that represents your primary and most frequent filter pattern, typically a date or region. For secondary filter columns, use Z-ordering instead. Z-ordering works within partitions and does not fragment data across directories. If your workload has multiple equally important filter columns with no clear primary, evaluate Liquid Clustering as an alternative to static partitioning. Liquid Clustering is more flexible and avoids the directory explosion problem entirely.

21

LAKEHOUSE DESIGN

Two teams are writing to the same Lakehouse. Data is randomly missing.

THE SITUATION

Two separate teams run pipelines that both write to Delta tables in a shared Lakehouse. Intermittently, rows that were confirmed written by one team are absent when the other team reads. No errors are thrown. The data appears to disappear randomly.

WRONG THINKING

Teams assume a Fabric platform bug or a storage issue. Since both pipelines succeed without errors and the missing data pattern is inconsistent, the problem looks like infrastructure flakiness rather than a concurrency design issue.

WHAT IS ACTUALLY HAPPENING

Both teams are performing full overwrites using `mode("overwrite")` without coordinating. When Pipeline A writes and then Pipeline B overwrites, Pipeline B replaces the entire table including any data Pipeline A just wrote. Delta Lake's ACID guarantees protect against partial writes but they do not protect against two concurrent full overwrites where each team assumes exclusive ownership. The last writer wins and everything written before it is gone.

THE FIX

Replace full overwrites with partition-level overwrites using `replaceWhere` so each team only owns and replaces its own partition. Alternatively, use `MERGE` for upsert patterns so writes are additive rather than destructive. For tables with multiple concurrent writers, establish clear ownership: each team writes to its own partition or its own table and a separate process consolidates if needed. Shared write access to the same table without coordination is a design problem, not a platform problem.

22

LAKEHOUSE DESIGN

You have been adding data daily for a year. Reads are getting progressively slower every month.

THE SITUATION

A Delta table that started fast has been getting measurably slower month over month. Nothing changed in the notebook. The data volume grew steadily but the slowdown is disproportionate to the growth. Queries that once took 30 seconds now take 8 minutes.

WRONG THINKING

The assumption is that more data means slower queries and the only fix is to archive old data or increase capacity. Engineers accept the degradation as an inevitable consequence of growth rather than recognising it as a maintenance failure.

WHAT IS ACTUALLY HAPPENING

Three compounding problems are building simultaneously without any maintenance running. First, the small files problem is getting worse with every daily append adding more tiny Parquet files. Second, the Delta transaction log is growing with hundreds of entries and checkpoint files that Spark must process before reading any data. Third, column statistics are increasingly stale as the data distribution evolves. Each of these problems individually is manageable. All three compounding over a year produces the disproportionate degradation.

THE FIX

Implement a scheduled maintenance notebook that runs after each daily load: `OPTIMIZE` with Z-ordering to compact files, `VACUUM` to remove stale files, and `ANALYZE TABLE` to refresh statistics. For the Delta log itself, ensure checkpointing is working correctly. Fabric Spark (based on Databricks Runtime 11.1+) creates a checkpoint every 100 commits by default; open-source Delta defaults to 10. Verify this is happening on your table using `DESCRIBE HISTORY` and adjust `delta.checkpointInterval` via `ALTER TABLE` if needed. Maintenance should be a first-class part of your data pipeline, not an afterthought.

23

LAKEHOUSE DESIGN

Your OneLake shortcut is fast in dev. Slow in production.

THE SITUATION

You created a shortcut in your Lakehouse pointing to data in another workspace or external storage. Dev queries through the shortcut are fast. When the same pattern is used in notebooks against larger data, performance is significantly worse than reading native Lakehouse tables.

WRONG THINKING

Shortcuts are presented as transparent access layers, meaning data accessed through a shortcut should behave the same as data stored natively. Engineers assume shortcut and native table performance should be equivalent and treat them interchangeably in notebooks.

WHAT IS ACTUALLY HAPPENING

The performance characteristics depend on what the shortcut points to. V-Order is applied at write time and inherits from the source — a shortcut to a Delta table written by Fabric to OneLake has V-Order; a shortcut to a Delta table written by Databricks or Synapse to ADLS Gen2 typically does not. Shortcut caching exists in Fabric but its coverage is uneven across source types: at the time of writing it supports shortcuts to S3, GCS, S3-compatible storage, and on-premises gateways, but ADLS Gen2 shortcuts are not in the cached set. A shortcut to external ADLS therefore makes a network call for every file metadata request and data read, where a native OneLake table reads from optimized local storage. In dev with small data the latency difference is negligible. At scale with thousands of file reads, the cumulative overhead becomes significant.

THE FIX

For ADLS-backed shortcuts in production read paths, copy the data into a native Lakehouse Delta table and run `OPTIMIZE` with V-Order to apply both compaction and the layout the read engine reads efficiently. For S3, GCS, or on-premises gateway shortcuts, enable shortcut caching in the workspace OneLake settings — this is the closest thing to native performance for those source types and is a real option, just not currently for ADLS. Use shortcuts for data exploration and federation patterns where latency tolerance is higher, and reserve native tables for production workloads that require consistent, high-performance reads at scale.

24

LAKEHOUSE DESIGN

You migrated from CSV to Parquet. Query performance barely changed.

THE SITUATION

A team migrated a large dataset from CSV files in OneLake to Parquet format expecting a significant read performance improvement. After migration, queries are marginally faster but not by the order of magnitude that was expected. The improvement does not justify the migration effort.

WRONG THINKING

Parquet is known to be far more efficient than CSV for analytical workloads, so engineers expect the format change alone to deliver dramatic improvements. When it does not, they assume the benchmark was wrong or that Fabric does not benefit from Parquet the same way other platforms do.

WHAT IS ACTUALLY HAPPENING

Parquet delivers its performance benefits through columnar storage, compression, and predicate pushdown via row group statistics. But raw Parquet files without Delta Lake wrapping have no file-level statistics that Spark can use for file skipping. The files are also not V-Order optimized, which is Fabric's proprietary sort order that significantly accelerates reads in the Fabric engine. Converting to Parquet was the right direction but stopping at raw Parquet captures only a fraction of the available performance benefit.

THE FIX

Convert from raw Parquet to Delta format by registering the files as a Delta table or rewriting them using `df.write.format("delta")`. Then run `OPTIMIZE` to apply V-Order and compact files. Delta wrapping adds the transaction log and statistics layer that Spark needs for file skipping and efficient planning. The combination of Delta format plus V-Order plus correct partitioning and Z-ordering is what delivers the full performance benefit on Fabric, not format conversion alone.

25

LAKEHOUSE DESIGN

Your Lakehouse has 3 years of data. Only the last 30 days matter. It is still slow.

THE SITUATION

A Delta table holds 3 years of historical data. Every production query filters for the last 30 days. Despite the filter, query times are far longer than they should be for 30 days of data. The table was partitioned by date when it was created.

WRONG THINKING

Since the table is partitioned by date and queries filter by date, engineers assume partition pruning is doing its job and only 30 days of files are being read. When performance is still poor, they attribute it to the overall table size rather than investigating what Spark is actually scanning.

WHAT IS ACTUALLY HAPPENING

Partition pruning is working and Spark is only reading 30 partitions. But each of those 30 date partitions has accumulated hundreds of small files from daily appends over 3 years of overlapping writes to recent partitions. The file count within the active partitions is the real problem. Additionally, the Delta transaction log for a 3-year-old table with daily writes has thousands of entries that Spark processes during query planning before a single data file is read. The bottleneck is metadata overhead, not data volume.

THE FIX

Run `OPTIMIZE` specifically on the active recent partitions using a `WHERE` clause to target only the last 60 to 90 days rather than the entire table. This compacts the high-activity partitions without the cost of touching 3 years of historical data. Verify Delta log checkpointing is current using `DESCRIBE HISTORY`. For tables older than 1 to 2 years with heavy query patterns on recent data only, evaluate whether archiving older partitions to a separate cold storage table reduces planning overhead meaningfully.

26

LAKEHOUSE DESIGN

Two Lakehouses hold the same data. Query speeds are completely different. Nobody knows why.

THE SITUATION

Two Lakehouses in different workspaces were created from the same source data at the same time. One consistently returns query results in 20 seconds. The other takes 4 minutes for the same query. Schema and row counts are identical.

WRONG THINKING

Engineers compare capacity configurations between the two workspaces first. When those look similar they assume a Fabric platform inconsistency or a workspace-level setting nobody has changed. The investigation goes in circles because the real difference is inside the tables themselves.

WHAT IS ACTUALLY HAPPENING

The fast Lakehouse was loaded using a process that wrote data with V-Order enabled and then ran OPTIMIZE on every table. The slow Lakehouse was loaded by copying raw Parquet files or by writing with a process that had V-Order disabled. V-Order is a Fabric-specific write-time sort optimization that dramatically accelerates reads in the Fabric engine. Tables without V-Order applied look identical in schema and row count but perform very differently because the physical layout of data within the Parquet files is completely different.

THE FIX

Verify V-Order status on both tables using `DESCRIBE DETAIL tablename` and checking the properties field. If V-Order is not applied, run `OPTIMIZE` on the slow table. OPTIMIZE in Fabric applies V-Order by default. Ensure any data loading pipelines have V-Order enabled at the session level with `spark.conf.set("spark.sql.parquet.vorder.enabled", "true")` set in the first cell of your notebook before any writes. V-Order is a write-time optimization so it must be applied either during the initial write or via a subsequent OPTIMIZE run.

27

LAKEHOUSE DESIGN

You gave a team read access to your Lakehouse. Your own jobs started slowing down.

THE SITUATION

You granted a second team read access to tables in your Lakehouse. Within a few days your own scheduled notebooks start running slower than before. No changes were made to your notebooks, tables, or data volumes. The slowdown correlates with the new team starting to use the access.

WRONG THINKING

Since read access was granted and reads are supposed to be non-interfering, engineers assume the correlation is coincidental. They investigate their own notebooks for changes and check table health before considering that the shared access is the cause.

WHAT IS ACTUALLY HAPPENING

Both teams share the same Fabric capacity units in the workspace. The second team's read queries, even though they are just reads, consume capacity units from the same shared pool. When both teams run heavy workloads simultaneously, total capacity demand exceeds available units and Fabric throttles or queues jobs from both teams. This is not a read versus write conflict at the storage level. It is a capacity contention problem at the workspace level. OneLake handles concurrent reads fine but the Spark compute layer is shared.

THE FIX

Review workspace capacity utilization metrics in the Fabric Monitoring Hub during the periods when slowdowns occur. If capacity utilization is consistently high, consider separating workloads across different workspaces or different capacity allocations. Establish usage windows where heavy workloads from different teams are staggered rather than concurrent. For the second team, consider providing them a dedicated copy of the data they need in their own workspace rather than sharing capacity on yours.

04

FABRIC ENGINEERING PATTERNS

Chapter 4 Session and Concurrency

Eleven scenarios covering how Spark sessions, multi-notebook concurrency, and Fabric capacity interact in ways that are invisible until something breaks.

28

SESSION AND CONCURRENCY

Your notebook reads the same large Delta table four times in different cells. Performance is terrible.

THE SITUATION

A notebook reads a large Delta table at the top, then reads it again in three subsequent cells with different filters for different transformations. Each read is a fresh scan. Total notebook runtime is much longer than expected given what each individual step does.

WRONG THINKING

Engineers treat each cell as independent and assume Spark is smart enough to reuse data it already read. Since the table is in OneLake and OneLake is fast, multiple reads feel like they should be cheap. The cumulative cost of repeated full scans is not obvious when looking at individual cells.

WHAT IS ACTUALLY HAPPENING

Spark DataFrames are lazy. Each time you call an action on a DataFrame that traces back to the same table read, Spark re-executes the entire read from OneLake. There is no automatic caching of source data between cells. A notebook that reads a 50GB table four times is scanning 200GB of data from OneLake even though logically it only needs to read 50GB once. This is one of the most common and easiest-to-fix sources of wasted compute in notebooks.

THE FIX

Read the table once into a DataFrame, then call `.cache()` or `.persist()` on it before the first action. All subsequent operations in the notebook will use the cached version in memory instead of re-reading from OneLake. Call `.unpersist()` when the DataFrame is no longer needed to free memory for subsequent stages. Only cache DataFrames that are genuinely reused multiple times. Caching a DataFrame that is used only once wastes memory without any benefit.

29

SESSION AND CONCURRENCY

You repartitioned your DataFrame to 1 partition before writing. The write took forever.

THE SITUATION

To avoid the small files problem you added `repartition(1)` before writing a large DataFrame to a Delta table. The idea was to produce a single output file. Instead the write took significantly longer than a normal write and the session appeared to stall.

WRONG THINKING

Writing to one file sounds like it should be simpler and faster than writing to many files. Less output means less work. This is the wrong mental model for how distributed Spark writes actually work.

WHAT IS ACTUALLY HAPPENING

`repartition(1)` forces all data across every executor in the session to be shuffled to a single partition on a single executor before writing. For a large DataFrame, this means moving potentially gigabytes of data across the network to one place, then writing it all from one executor sequentially. The write itself is now completely serial. You have intentionally destroyed all parallelism in your distributed system to produce one file. The small files problem is real but this is not the correct solution for large data.

THE FIX

Use `coalesce(n)` instead of `repartition(1)` where n produces output files of roughly 128MB to 256MB each. `coalesce` reduces partition count without a full shuffle by merging existing partitions locally. For Delta tables, the better long-term solution is to write normally and let OPTIMIZE compact the files afterwards on a schedule. Never use `repartition(1)` on large DataFrames in production. Reserve single-file output only for small reference datasets where the file size genuinely warrants it.

30

SESSION AND CONCURRENCY

You have 10 transformations chained. The session crashes halfway through with no clear error.

THE SITUATION

A notebook with a long chain of DataFrame transformations crashes partway through execution. The error message is unhelpful, often a generic executor lost or task failed message. The failure point changes between runs. Sometimes it crashes at step 6, sometimes at step 9.

WRONG THINKING

Engineers focus on the step where the error appears and spend time debugging that specific transformation. Since the failure point moves between runs, they assume Fabric session instability or a platform bug rather than looking at the accumulated execution plan.

WHAT IS ACTUALLY HAPPENING

Spark's lazy evaluation means that chaining 10 transformations without any intermediate actions builds up a single enormous execution plan. When a terminal action is finally called, Spark attempts to execute the entire chain in one job. The plan becomes very large, the lineage graph is deep, and the memory required to track and execute all stages simultaneously strains the session. The failure point varies because memory pressure accumulates differently depending on data distribution on that run. This is sometimes called lineage explosion.

THE FIX

Break the transformation chain into logical checkpoints by calling `.cache()` followed by a lightweight action like `.count()` at strategic intermediate points. This forces Spark to materialize and store the result at that point, truncating the lineage graph. Subsequent transformations build a new, shorter plan from the cached result. This pattern trades some memory for dramatically improved plan manageability and session stability. Aim to checkpoint after every 3 to 4 heavy transformations in long notebooks.

31

SESSION AND CONCURRENCY

One heavy notebook is running. Everyone else in the workspace slows down.

THE SITUATION

When a particular heavy notebook runs, other team members report that their notebooks, reports, and queries become noticeably slower. As soon as the heavy notebook finishes, everyone else returns to normal. The heavy notebook itself runs fine in isolation.

WRONG THINKING

Since the heavy notebook is a separate session and Spark sessions are supposed to be isolated, engineers assume the impact on others is coincidental or caused by something else. The concept of session isolation leads people to believe one notebook cannot affect another.

WHAT IS ACTUALLY HAPPENING

Spark sessions in Fabric are isolated at the execution level but they share the same Fabric capacity pool in the workspace. The heavy notebook is consuming a large portion of available capacity units, leaving less for everyone else. Fabric throttles other workloads to protect overall workspace stability. This is not a Spark isolation failure. It is capacity contention. The workspace has a fixed capacity budget and a single heavy consumer can crowd out all other workloads.

THE FIX

Use the Fabric Monitoring Hub to identify peak capacity consumption periods and which workloads are responsible. Schedule heavy notebooks during off-peak hours or in a dedicated workspace with its own capacity allocation so it cannot impact shared workloads. Review whether the heavy notebook can be broken into lighter stages that release capacity between runs. For workloads that must run during business hours, work with workspace administrators to review capacity sizing relative to concurrent workload demand.

32

SESSION AND CONCURRENCY

Your job runs fast alone. It runs slowly when 3 other jobs run at the same time.

THE SITUATION

A notebook consistently completes in 8 minutes when run in isolation. When the same notebook runs concurrently with 3 other scheduled notebooks, runtime jumps to 35 minutes or more. Each notebook is independent with no shared state or dependencies.

WRONG THINKING

Because the notebooks are independent and share no data or code, engineers assume concurrent execution should have no effect on individual runtimes. The slowdown feels like a platform bug rather than an expected consequence of shared resource consumption.

WHAT IS ACTUALLY HAPPENING

All four notebooks are drawing from the same Fabric capacity pool simultaneously. Each notebook requests compute resources at session startup and during heavy stages. When four sessions all hit heavy stages at the same time, total capacity demand exceeds available units. Fabric queues tasks rather than failing them, which is why the jobs complete but take much longer. The 8-minute solo runtime is the compute time. The 35-minute concurrent runtime includes both compute time and queue wait time.

THE FIX

Stagger notebook start times so heavy stages do not overlap. Even a 5 to 10 minute offset between scheduled starts can significantly reduce peak concurrent capacity demand. Review the Fabric Monitoring Hub smoothed CU chart to understand your workspace capacity usage pattern across the day. For critical time-sensitive notebooks, schedule them during the lowest utilization window. For non-critical notebooks, add deliberate delays to keep them out of peak periods.

33

SESSION AND CONCURRENCY

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

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.

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.

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 (the same partition or the same files), 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. This is by design. ACID guarantees consistency but does not mean unlimited transparent concurrency.

THE FIX

The right fix depends on whether the table is partitioned. For **partitioned tables**, design the two pipelines to write to non-overlapping partitions so their commits never conflict. For **unpartitioned tables** (most tables by default), partition advice does not apply — the real answer is serialization: use pipeline dependencies, scheduling offsets, or a control table to ensure only one pipeline writes at a time. As a last resort, use MERGE with a deterministic key so writes are additive rather than destructive. Add retry logic with exponential backoff to handle `ConcurrentWriteException` gracefully in cases where occasional overlap is unavoidable. This scenario duplicates Book I Scenario 5 from a Spark-engineer perspective. Never design two independent pipelines to write to the same Delta partition concurrently without a coordination mechanism.

34

SESSION AND CONCURRENCY

Your pipeline runs fine at 1 million rows. It falls over at 100 million rows with a different failure every time.

THE SITUATION

A pipeline that processes and writes data to a Delta table works perfectly at 1 million rows in testing. At full production volume of 100 million rows it fails, but the error and the stage of failure are different on each attempt, making it hard to diagnose consistently.

WRONG THINKING

The inconsistent failure point leads engineers to assume Fabric infrastructure instability. Since the same code works at small scale and no single stage fails consistently, the problem feels environmental rather than structural. Each run gets investigated independently without finding a common root cause.

WHAT IS ACTUALLY HAPPENING

At 100 million rows, multiple interacting problems manifest simultaneously. Shuffle partitions are too few, causing some tasks to be massive and spill to disk. Data skew in join or groupby keys means certain partitions are orders of magnitude larger than others. The combination of spill and skew creates memory pressure that pushes different stages past their limits on different runs depending on which tasks happen to run on which executor slots in that session. The non-determinism comes from the interaction of these multiple problems, not from one specific bug.

THE FIX

Treat this as a systematic investigation rather than a single bug hunt. First increase shuffle partitions substantially and re-run to see if the failure point stabilises. Then check data distribution on every join and groupby key to identify skew. Fix the most severe skew using salting or explicit handling of dominant keys. Add intermediate `.cache()` calls after heavy stages to truncate lineage and stabilise memory usage. Resolve each problem layer independently before combining fixes and validating on full production volume.

35

SESSION AND CONCURRENCY

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

THE SITUATION

A daily incremental load appends roughly the same number of rows to a Delta table on 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 incremental volume has not changed.

WRONG THINKING

Since the incremental volume is constant, engineers assume the load time should also be constant. When it grows they look at the new data for anomalies, check network throughput to OneLake, and review source system performance. None of these are the cause.

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 thousands of daily appends without compaction, the table has tens of thousands of small files. Before every write, Spark lists all the active files 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 bound this cost — when checkpoints are firing as expected, log size is not the bottleneck. When they are not, the log read cost grows over time. This scenario duplicates Book I Scenario 1 from a Spark-engineer perspective; the underlying cause and fix are the same.

THE FIX

Address file count first because it is the dominant cause. Run `OPTIMIZE` on the table to compact small files, then `VACUUM` with appropriate retention to remove obsolete files. Schedule both as recurring maintenance. 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, set `delta.checkpointInterval` via `ALTER TABLE`. For tables with very long histories where the working set is recent data, consider moving older partitions to a separate archive table.

36

DELTA LAKE

You enabled column statistics collection. Write performance dropped significantly.

THE SITUATION

Column statistics were enabled on a wide Delta table to improve query planning. Reads improved as expected. But write performance dropped noticeably, adding several minutes to every daily load. The table has over 300 columns.

WRONG THINKING

Statistics collection is assumed to be a background or lightweight operation. Engineers enable it globally across all columns expecting only read benefits without considering the write cost, especially on wide tables.

WHAT IS ACTUALLY HAPPENING

When column statistics are enabled, Delta computes min, max, null count, and distinct count for each tracked column on every write. For a table with 300 columns, this means computing statistics for 300 columns across every row group in every file written. The compute cost scales with column count and write volume. On narrow tables this overhead is negligible. On wide tables it becomes a meaningful fraction of total write time, especially when all columns are tracked rather than just the columns used in query filters.

THE FIX

Two table properties give you control. `delta.dataSkippingNumIndexedCols` is a **count** that limits stats collection to the first N columns in the schema (defaults to 32) — useful if your filter columns sit at the front of the schema. `delta.dataSkippingStatsColumns` takes an explicit **list** of column names to collect stats on, regardless of position — useful for wide tables where filter columns are scattered. For a 300-column table where queries only filter on 5 columns, use `dataSkippingStatsColumns` with the names of those 5 columns; you'll get the read benefit without paying for stats on the other 295. After changing either property, run `ANALYZE TABLE table_name COMPUTE DELTA STATISTICS` to recompute stats for existing data. Review which columns appear in WHERE clauses and JOIN conditions before deciding which property to use.

37

DELTA LAKE

You restored a Delta table to a previous version. Downstream tables are now inconsistent.

THE SITUATION

A bad data load corrupted a source Delta table. You used time travel to restore it to the last known good version. The source table looks correct. But two downstream Gold tables that were built from it are now out of sync with each other and with the restored source.

WRONG THINKING

Restoring the source table feels like a complete fix. Engineers assume that once the source is correct, everything downstream will be correct on the next pipeline run. The restore is treated as an isolated operation rather than the start of a broader remediation process.

WHAT IS ACTUALLY HAPPENING

Delta time travel restores the source table but it does not touch anything downstream. The downstream Gold tables were built from the corrupted version and still contain corrupted or excess data. On the next pipeline run, incremental logic only processes new data since the last watermark, which means the corrupted rows already written to the Gold tables will never be corrected unless the downstream tables are explicitly rebuilt or the affected partitions are overwritten. The source is clean but the derived state is still wrong.

THE FIX

After restoring a source table, identify every downstream table that consumed data from the corrupted version by checking pipeline run timestamps against the corruption window. For each affected downstream table, overwrite the affected partitions by rerunning the pipeline with a full reload for that time range rather than an incremental load. Document the restore action and the downstream remediation steps. A source restore without downstream remediation is an incomplete fix.

38

DELTA LAKE

You dropped and recreated a Delta table with the same name. Downstream shortcuts broke silently.

THE SITUATION

A Delta table was dropped and recreated with the same name as part of a schema migration. The table looks identical from the outside. But other teams report that their Lakehouse shortcuts pointing to that table are returning empty results or stale data.

WRONG THINKING

Since the table name is identical and the schema is the same, engineers assume shortcuts will continue working transparently. The drop and recreate feels like a non-breaking change because the external interface looks unchanged.

WHAT IS ACTUALLY HAPPENING

OneLake shortcuts point to a specific physical path in storage, not to a logical table name. When a Delta table is dropped and recreated, a new storage path is created. The old path still exists until VACUUM removes it, or it may already be empty. The shortcut is still pointing to the old path, which is either empty or contains stale pre-migration data. The table name match is irrelevant because shortcuts operate at the storage path level, not the metadata level.

THE FIX

Before dropping any Delta table that other teams have shortcuts to, notify all shortcut owners so they can update their references after the recreate. After recreating the table, share the new physical path so shortcuts can be updated. For critical shared tables, prefer schema evolution using `ALTER TABLE` over drop and recreate to avoid path changes entirely. Maintain a registry of which tables have external shortcuts so the impact of any table recreation can be assessed before it happens.

39

DELTA LAKE

Your Delta table has 200 columns but only 5 change on every write. Writes are still slow.

THE SITUATION

A wide Delta table receives daily updates to only 5 columns out of 200. The update logic uses MERGE targeting just those 5 columns. Despite the narrow update scope, write times are disproportionately long relative to the rows being changed.

WRONG THINKING

Since only 5 columns are being updated, engineers expect the write to be fast regardless of total column count. The assumption is that Delta is smart enough to only rewrite the changed columns. This is a reasonable assumption but it is wrong for how Delta actually works.

WHAT IS ACTUALLY HAPPENING

Delta Lake uses copy-on-write at the file level, not the column level. When a MERGE updates 5 columns in a row, Delta rewrites the entire Parquet file containing that row, including all 200 columns. There is no partial column write. Every file touched by the MERGE is fully rewritten with all columns. On a wide table, each file rewrite is expensive because it must read, deserialise, modify, reserialise, and write all 200 columns even though only 5 changed. Column count directly multiplies the cost of every file rewrite.

THE FIX

For tables where a small subset of columns changes frequently while the rest are static, consider splitting the table into a narrow frequently-updated table and a wide rarely-updated table joined on a primary key. Updates only touch the narrow table, which rewrites far less data per file. Alternatively, evaluate whether the 200-column design is genuinely necessary or whether vertical decomposition into domain-specific tables better reflects the actual access and update patterns of the data.

40

SPARK PERFORMANCE

Your job spills to disk on every run despite the session appearing to have plenty of memory.

THE SITUATION

The Spark UI consistently shows spill to disk metrics on a heavy aggregation stage. Total session memory appears adequate based on the session configuration. The job completes but is significantly slower than it should be due to disk spill on every run.

WRONG THINKING

Since total session memory looks sufficient on paper, engineers assume the spill is caused by a brief memory spike that cannot be avoided. They accept the spill as a normal part of the job rather than investigating why memory is insufficient at the task level despite being sufficient at the session level.

WHAT IS ACTUALLY HAPPENING

Spark divides session memory across executor slots and further divides each slot between execution memory and storage memory. If shuffle partitions are too few, individual partition sizes are large and a single task may need more execution memory than its allocated share of one executor slot. Even if the total session has 32GB, a single task is limited to its fraction of one slot. Spill happens at the task level not the session level. Total memory is not the constraint. Per-task memory allocation is.

THE FIX

Increase `spark.sql.shuffle.partitions` to reduce the amount of data each individual task must hold in memory. Check the Spark UI spill metrics on the affected stage and look at the shuffle read size per task. Aim for tasks handling no more than 200MB to 300MB of shuffle data. Additionally check `spark.memory.fraction` and `spark.memory.storageFraction` if you have cached DataFrames competing for memory with execution. Reducing cache footprint can free execution memory and eliminate spill without increasing total session size.

41

SPARK PERFORMANCE

You increased shuffle partitions to 2000. The job got slower not faster.

THE SITUATION

A job was suffering from large shuffle partitions causing spill. Shuffle partitions were increased from 200 to 2000 expecting improved performance. Instead the job became slower. Task count exploded and the Spark UI showed thousands of very small short-lived tasks.

WRONG THINKING

More shuffle partitions means smaller partitions means less spill means better performance. This logic is correct up to a point but engineers often increase partition count without considering the overhead that comes with managing thousands of tiny tasks.

WHAT IS ACTUALLY HAPPENING

Every Spark task carries fixed overhead regardless of how much data it processes: task scheduling, serialisation, deserialisation, and result reporting back to the driver. When partitions become very small, this fixed overhead becomes a larger proportion of total task time than actual computation. At 2000 partitions on a moderately sized dataset, each task may process only a few kilobytes of data but still incur the full task scheduling cost. The overhead of managing 2000 tasks exceeds the benefit of smaller partitions. This is the task overhead trap.

THE FIX

Target shuffle partition sizes of 128MB to 256MB per partition rather than targeting a specific partition count. Calculate the appropriate count by dividing total shuffle data size by your target partition size. For a 100GB shuffle, 400 to 800 partitions is a reasonable range. Use AQE's `spark.sql.adaptive.coalescePartitions.enabled` setting to let Spark automatically merge small post-shuffle partitions at runtime, which handles cases where data distribution is uneven across partitions.

42

SPARK PERFORMANCE

Two identical DataFrames joined together. The result has unexpected duplicate rows.

THE SITUATION

A join between two DataFrames produces more rows than expected. Spot-checking the output reveals duplicate rows for certain key values. Both source DataFrames look clean. The join condition looks correct. The duplicates appear only for specific key values, not uniformly.

WRONG THINKING

Engineers check the join condition for bugs and recount rows in both source DataFrames. When source counts look right and the join condition looks correct, the duplicates are attributed to a Spark bug or an unexpected platform behaviour. The investigation focuses on the join itself rather than the source data.

WHAT IS ACTUALLY HAPPENING

The source DataFrames contain duplicate keys. A join does not deduplicate. It 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. The duplicates in the source are not visible when doing a simple row count because total row count may still look correct if other keys are underpopulated. The join is working correctly. The source data is not.

THE FIX

Before any join, validate key uniqueness in both DataFrames by comparing `df.count()` against `df.select(key_col).distinct().count()`. If these differ, duplicates exist in the source and must be resolved before joining. Decide whether to deduplicate using `dropDuplicates([key_col])` or whether the duplicates are meaningful and the join type needs to change. Add key uniqueness assertions as standard data quality checks at the start of notebooks that perform joins on business keys.

43

SPARK PERFORMANCE

Your explain plan looks efficient. The actual runtime does not match the plan at all.

THE SITUATION

You ran EXPLAIN on a query before executing it. The plan showed a broadcast join and file pruning that should make the query fast. You run the actual query and it is 20 times slower than the plan suggested. The Spark UI shows a sort-merge join instead of a broadcast join.

WRONG THINKING

Engineers trust EXPLAIN output as a reliable preview of actual execution. If the plan shows a broadcast join, they expect a broadcast join to execute. The disconnect between plan and execution is unexpected and feels like inconsistent platform behaviour.

WHAT IS ACTUALLY HAPPENING

EXPLAIN shows the logical or optimised plan based on statistics available at planning time. With AQE enabled, Spark can change the physical execution plan at runtime based on actual shuffle sizes and partition statistics observed during execution. If at runtime the data turns out larger than statistics predicted, Spark may switch from a broadcast join to a sort-merge join. The EXPLAIN plan was correct at planning time. The runtime plan reflects what actually happened with real data. These two plans are legitimately different when AQE is active.

THE FIX

Use `df.explain(mode="formatted")` in PySpark or `EXPLAIN FORMATTED` in Spark SQL to see the actual runtime plan rather than the estimated plan. Always review the Spark UI SQL tab after execution to confirm what plan was actually used. For critical queries where you need to guarantee a specific join strategy, use explicit hints like `broadcast()` to override AQE decisions. Keep table statistics fresh with `ANALYZE TABLE` to reduce the gap between estimated and actual plan. Never use pre-execution EXPLAIN as the sole basis for performance assumptions on large tables with stale statistics.

44

LAKEHOUSE DESIGN

You built a Gold table to serve dashboards. It is slower to query than the Silver table it came from.

THE SITUATION

A Gold layer table was created by aggregating and joining Silver tables. It is smaller in row count and was designed specifically for Power BI consumption. But direct queries on the Gold table are slower than equivalent queries on the underlying Silver tables.

WRONG THINKING

A smaller, pre-aggregated table should always be faster than a larger raw table. Engineers assume the Gold table will be the fastest read path by definition and are surprised when it underperforms. The assumption is that less data always means faster queries.

WHAT IS ACTUALLY HAPPENING

The Gold table was written without OPTIMIZE being run afterwards. It exists as many small files from the aggregation write process. The Silver tables by contrast have been maintained with regular OPTIMIZE runs and have V-Order applied. A well-maintained Silver table with V-Order and proper file sizes will consistently outperform a Gold table that is smaller in rows but fragmented into hundreds of tiny unoptimised files. File health matters more than row count for read performance in Fabric.

THE FIX

Run `OPTIMIZE` on the Gold table immediately after every write to apply V-Order and compact files. Add Z-ordering on the columns most commonly used in Power BI filters, typically date and category columns. Make Gold table maintenance part of the same pipeline that builds it, not a separate scheduled job that may be missed. A Gold table without post-write OPTIMIZE is not a serving layer. It is an unoptimised intermediate result that happens to have fewer rows.

45

LAKEHOUSE DESIGN

Your notebook writes to OneLake. Reading back immediately returns stale or missing data.

THE SITUATION

A notebook writes a DataFrame to a Delta table and then reads it back in the next cell to validate the write. The read returns fewer rows than were written, missing columns, or data from a previous version. The write appeared to complete successfully with no errors.

WRONG THINKING

A successful write followed immediately by a read should always return the just-written data. Engineers assume the write is fully committed and visible before the read starts, the same way a database transaction would behave. When stale data appears they suspect a write failure or a Fabric storage bug.

WHAT IS ACTUALLY HAPPENING

The write succeeded and the data is in OneLake. But the read is opening a cached or previously resolved version of the Delta table metadata rather than reading the latest committed version. This can happen when the DataFrame used for reading was created before the write completed, when Delta table caching is returning a stale snapshot, or when the notebook reuses a table reference that was resolved at session startup. Delta guarantees consistency for committed writes but only if the reader resolves the table version after the commit.

THE FIX

Always create a fresh DataFrame reference after a write rather than reusing one created before the write. Call `spark.read.format("delta").load(table_path)` or use `spark.table(table_name)` after the write completes to ensure the reader resolves the latest committed version. If using Delta table caching, call `DeltaTable.forName(spark, table_name).detail()` to confirm the latest version before reading. Never assume a DataFrame reference created before a write will reflect data written after it.

46

LAKEHOUSE DESIGN

You store JSON strings in a Delta column. Queries on nested fields are very slow.

THE SITUATION

A Delta table stores semi-structured data as JSON strings in a single string column. Queries that extract specific fields from those JSON strings using `get_json_object` or `from_json` are extremely slow, often scanning the entire table regardless of what filters are applied.

WRONG THINKING

Storing JSON in a string column feels like a pragmatic choice for flexible schema data. Engineers assume Spark handles JSON extraction efficiently since it is a built-in function. The performance cost of the storage design itself is not visible until query volume grows.

WHAT IS ACTUALLY HAPPENING

JSON strings stored in a single column are opaque to Spark's query planner. Parquet column statistics, predicate pushdown, and file skipping all operate on typed column values. None of these optimisations apply to a string column containing JSON. Every query must read and deserialise every row to extract the JSON field being filtered or selected. There are no file-level statistics for values inside a string. The entire table is a black box to the optimiser, so every query becomes a full scan regardless of the filter.

THE FIX

Explode the JSON structure into proper typed Delta columns using `from_json` with an explicit schema at write time rather than at read time. Typed columns benefit from Parquet statistics, predicate pushdown, and Z-ordering. For frequently queried nested fields, promote them to top-level columns. If full JSON flexibility is genuinely required, store the typed extracted fields alongside the raw JSON string so filters and statistics operate on the typed columns while the raw JSON remains available for edge cases.

47

LAKEHOUSE DESIGN

Your pipeline succeeded. The downstream table has duplicate rows.

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. The previous day's data has no duplicates.

WRONG THINKING

Since the pipeline succeeded, engineers look for duplicates in the source data first. When the source looks clean they assume the incremental logic has a bug that appeared suddenly. The pipeline infrastructure itself is rarely the first suspect when a job completes without error.

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 were appended again on the retry. The pipeline logic is not inherently buggy. The retry strategy is not safe for append-mode writes.

THE FIX

Design incremental pipelines to be idempotent by using MERGE instead of append, or by deleting the target partition before rewriting it in each run. Add a deduplication step at the start of any notebook that reads from a table written by a retryable pipeline. For append-mode pipelines, add a watermark check that prevents reprocessing already-loaded records. This scenario shares its core cause with Book I Scenario 4 (pipeline retry duplicates); the principle is the same. The rule is simple: any pipeline that can be retried must produce the same result whether it runs once or ten times.

48

SESSION AND CONCURRENCY

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

THE SITUATION

A notebook that typically runs in 4 minutes consistently takes 12 to 15 minutes on its first run each morning. Every subsequent run throughout the day takes the expected 4 minutes. Nothing changes between the first and second run. The data volume is the same.

WRONG THINKING

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

WHAT IS ACTUALLY HAPPENING

Fabric's serverless Spark sessions are fully deallocated when idle. 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 a single line of notebook code executes. Subsequent runs reuse the warmed session and the populated file cache, which is why they run at expected speed. The notebook code is identical. The session state is not.

THE FIX

Schedule a lightweight warm-up notebook to run 5 to 10 minutes before your main notebook each morning. The warm-up notebook simply reads a small amount of data from the same Lakehouse, which starts the session and begins warming the cache. Your main notebook then runs against an already-warm session. If your workload is notebook-heavy and you orchestrate multiple notebooks within a pipeline, enable High Concurrency Mode in the workspace settings — this packs multiple notebook activities 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 runs" feature. For standalone notebook runs, the warm-up pattern is the practical answer. This scenario duplicates Book I Scenario 2 from a Spark-engineer perspective.

49

SESSION AND CONCURRENCY

You enabled Delta caching. Memory usage spiked but query speed did not improve.

THE SITUATION

Delta caching was enabled expecting faster repeat reads on frequently queried tables. Memory consumption in the session increased noticeably. But query times on the same tables did not improve and in some cases became slightly worse due to memory pressure on other operations.

WRONG THINKING

Caching is assumed to universally improve performance. Enabling Delta caching feels like a free optimisation that should make all reads faster without any downside. The idea that caching could consume memory without delivering proportional read performance improvement is counterintuitive.

WHAT IS ACTUALLY HAPPENING

Delta caching in Fabric caches decompressed Parquet column data on local executor storage to speed up repeat reads of the same files. It delivers meaningful benefit only when the same files are read multiple times within a session. If each notebook run reads different partitions or if the notebook runs once and terminates, the cache is populated but never reused. The cache memory is consumed on the first read and the second read that would benefit from it never happens. Caching without repeated reads of the same data is pure overhead.

THE FIX

Enable Delta caching only for workloads with genuine repeated reads of the same data within a session or across closely spaced runs. Interactive exploration notebooks and Power BI Direct Lake queries are good candidates. Single-run batch notebooks that read each partition once are not. Before enabling caching broadly, verify the read pattern justifies it by checking whether the same files appear in multiple stages of the Spark UI. If they do not, caching adds memory pressure without benefit.

50

SESSION AND CONCURRENCY

Your notebook runs perfectly interactively. Scheduled runs fail consistently.

THE SITUATION

A notebook works without any issues when run manually in the Fabric UI. When the same notebook runs on a schedule it fails consistently, always at a different cell depending on the run, with permission errors, missing table errors, or session configuration failures.

WRONG THINKING

Since the notebook works interactively, engineers assume scheduled runs should behave identically. The difference between interactive and scheduled execution is not well understood and the failures feel random and platform-specific rather than design-related.

WHAT IS ACTUALLY HAPPENING

Interactive and scheduled notebook runs differ in three important ways. First, the identity context is different: interactive runs use your personal credentials while scheduled runs use a service principal or system identity that may not have the same permissions to Lakehouses, Key Vaults, or external connections. Second, session configuration set interactively through the UI is not persisted to scheduled runs unless set explicitly in code. Third, relative paths, hardcoded workspace references, and environment variables that resolve correctly in an interactive session may not resolve the same way in a scheduled context.

THE FIX

Set all Spark session configurations explicitly in the first cell of the notebook in code rather than relying on UI settings. Replace all hardcoded paths and workspace references with values read from notebook parameters or environment-aware configuration cells. Verify that the service principal or system identity used for scheduled runs has the necessary permissions to all Lakehouses, connections, and Key Vault references used in the notebook. Test scheduled behaviour by manually triggering a scheduled run and reviewing the run log before trusting the schedule in production.

CONCLUSION

The Lakehouse rewards the engineer who understands Delta.

Across 50 scenarios, one pattern repeats. Spark and Delta Lake behave differently in Fabric than engineers expect from standalone Spark experience. File layout determines query speed more than transformation logic. Session startup cost dominates short notebook runs. Concurrent writes without coordination corrupt tables silently. The engineers who build reliable Lakehouses are the ones who understand what Delta is doing underneath, not just what their code is doing on top.

ALSO IN THE SERIES

Pipelines and Data Factory

> Your pipeline succeeded. The target table is empty.
>
> Your trigger was active. It silently stopped firing months ago.

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.

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