Architecture Patterns

Common Patterns. Clear Fixes.

50 architecture patterns covering storage layer design, OneLake and shortcuts, Medallion architecture, workspace and capacity design, security architecture, CI/CD and environment strategy, and cost architecture in Microsoft Fabric. Patterns every Fabric engineer must recognise.

AUTHOR

Sanjay Chandra

50
SCENARIOS

INTRODUCTION

Architecture mistakes look reasonable until they do not.

Books I through IV of this series covered what breaks when you are building. This book covers what breaks when you are designing. The decisions made before writing a single pipeline activity or notebook cell determine whether the platform you build is maintainable, secure, cost-effective, and scalable. The wrong choices at the architecture layer create the exact problems covered in the earlier books.

The patterns in this book share a common characteristic. Every wrong decision described here was made by a reasonable engineer with a reasonable goal. Putting everything in one Lakehouse is simple and fast to start with. Using shortcuts everywhere avoids data duplication. Giving everyone Contributor access removes friction. Each of these decisions makes sense in isolation and at small scale. They all become serious problems as the platform grows.

This book covers seven chapters. Chapter 1 covers storage layer design: when to use Lakehouse, Warehouse, and SQL endpoint and what happens when the wrong choice is made. Chapter 2 covers OneLake design and shortcuts, where convenience frequently becomes technical debt. Chapter 3 covers Medallion architecture anti-patterns that only appear after the platform has been running for months. Chapter 4 covers workspace and capacity design. Chapter 5 covers security architecture where the gap between apparent security and actual security is widest. Chapter 6 covers CI/CD and environment strategy. Chapter 7 covers cost architecture, where the most expensive mistakes are the ones that look like optimizations.

Each scenario follows the same structure as the rest of the series: the situation, the wrong thinking, what is actually happening, and the fix. Architecture patterns take longer to manifest than pipeline errors or query slowdowns. But when they do manifest, they affect everything built on top of them.

01

FABRIC ENGINEERING PATTERNS

Chapter 1 Storage Layer Design

Eight scenarios covering when to use Lakehouse, Warehouse, and the SQL analytics endpoint and what happens when the wrong storage layer is chosen for the wrong workload.

01

STORAGE LAYER DESIGN

You put all data in a Lakehouse. Your SQL-heavy team is constantly complaining about query performance.

THE SITUATION

An engineering team built the entire Fabric platform on Lakehouses. Data engineers are happy. But the analytics team running complex multi-table SQL joins, window functions, and stored procedures reports consistently poor performance and missing SQL features. They are asking to move back to a traditional database.

WRONG THINKING

Engineers assume the Lakehouse SQL endpoint provides equivalent SQL performance to a dedicated Warehouse. Since both accept T-SQL queries, the two are treated as interchangeable. The performance gap is attributed to the SQL queries being poorly written rather than to the storage layer choice.

WHAT IS ACTUALLY HAPPENING

The Lakehouse SQL analytics endpoint is optimized for reading Delta files through a translation layer. It supports a subset of T-SQL and its query planner is different from the Fabric Warehouse MPP engine. Complex analytical SQL workloads involving multi-table joins with statistics-driven plans, stored procedures, and fine-grained workload management benefit significantly from the Warehouse engine. The Lakehouse endpoint was designed for ad-hoc exploration and light reporting, not as a primary SQL serving layer for heavy analytical workloads. Putting SQL-heavy workloads on a Lakehouse endpoint and expecting Warehouse-level performance is a fundamental layer mismatch.

THE FIX

Move SQL-heavy serving workloads to a Fabric Warehouse. Keep the Lakehouse for raw and transformed Delta storage and use the Warehouse as the semantic layer for analytical SQL. Load data from the Lakehouse into the Warehouse using pipelines or COPY INTO for tables that require Warehouse-level query performance. This separation allows each layer to do what it does best: the Lakehouse handles flexible Delta storage and Spark transformations, the Warehouse handles optimized SQL serving.

02

STORAGE LAYER DESIGN

You moved everything to Fabric Warehouse. Your data science team can no longer run Spark notebooks on the data.

THE SITUATION

After performance complaints about the Lakehouse SQL endpoint, the team migrated all data into Fabric Warehouse tables. SQL performance improved significantly. But the data science team that was running PySpark notebooks for feature engineering, model training, and exploratory analysis can no longer access the data the same way. Their notebooks either fail or require complex workarounds.

WRONG THINKING

Engineers assume that since all data is now in Fabric, the data science team should be able to access it through some mechanism. The fundamental incompatibility between Warehouse-native tables and Spark-based access is not considered during the migration planning.

WHAT IS ACTUALLY HAPPENING

Fabric Warehouse tables are stored in a columnar format managed by the Warehouse engine. They are not Delta tables in OneLake and cannot be read directly by Spark sessions the way Lakehouse Delta tables can. A data science team that reads data using `spark.read.format('delta')` or accesses tables through the Spark catalog cannot use Warehouse-native tables through the same path. The Warehouse and the Spark execution engine are separate layers. Moving everything to Warehouse solves the SQL performance problem but eliminates the Spark access that data science workflows depend on.

THE FIX

Design the platform with two serving layers: a Lakehouse holding Delta tables for Spark-based workloads including data science, ML, and PySpark transformations, and a Warehouse for SQL-heavy analytical serving. Use pipelines to keep the Warehouse in sync with the curated Lakehouse Gold layer. Each team accesses the layer designed for their tooling. Data scientists read from the Lakehouse, SQL analysts query the Warehouse. Neither layer needs to serve both workloads simultaneously.

03

STORAGE LAYER DESIGN

You are querying Lakehouse data through the SQL endpoint. Results differ from querying the same data through a notebook.

THE SITUATION

A data engineer validates pipeline output by querying the target Lakehouse table through a notebook and gets the expected result. A downstream analyst queries the same table through the SQL analytics endpoint and gets a different row count or different aggregated values. Both queries are running against the same Lakehouse. Neither result is obviously wrong.

WRONG THINKING

Engineers assume the SQL endpoint and the notebook are reading identical data because they reference the same Lakehouse table. The difference in results is attributed to a data quality issue or a user error rather than to the two access paths reading different versions of the data.

WHAT IS ACTUALLY HAPPENING

The Lakehouse SQL analytics endpoint uses a metadata synchronization process that periodically reflects the current Delta table state. The Spark notebook reads the Delta transaction log directly and always sees the latest committed version. If a pipeline wrote data after the last SQL endpoint sync, the notebook sees the new data while the SQL endpoint serves the previous version. This sync lag means the two access paths can return different results for a window of time after every data write. The gap is usually minutes but can be longer during periods of high workspace activity.

THE FIX

Use a single access path for any validation that requires consistency. For post-load validation in pipelines, use notebook activities or Spark SQL that read the Delta table directly rather than queries through the SQL endpoint. Document the sync lag behavior for downstream consumers so they understand that SQL endpoint results may lag behind actual data by a predictable window. For time-sensitive reporting that must reflect the latest committed data, connect Power BI through the Warehouse endpoint or a refreshed semantic model rather than through the SQL analytics endpoint.

04

STORAGE LAYER DESIGN

You built your serving layer in the Lakehouse. Power BI reports are slow despite the data being small.

THE SITUATION

A Gold layer was built in the Lakehouse with pre-aggregated summary tables specifically designed for Power BI consumption. The tables are small, containing only a few hundred thousand rows. Despite the small data size, Power BI reports connecting through the SQL analytics endpoint are slow and Power BI Direct Lake performance does not meet expectations.

WRONG THINKING

Engineers optimize the Gold table design and add more pre-aggregations. Performance improves slightly but not to the level expected for data of this size. The serving layer architecture itself is not reconsidered because the Lakehouse seems like the natural place for Gold data.

WHAT IS ACTUALLY HAPPENING

For Power BI serving, the optimal path depends on the connection mode. Direct Lake mode on Lakehouse performs best when Delta tables are well-optimized with V-Order applied and compact file sizes. If the Gold tables were written without OPTIMIZE running afterwards, V-Order is not applied and Direct Lake framing is inefficient. For SQL endpoint connections, the translation layer adds latency that becomes noticeable even on small tables. A Fabric Warehouse serving layer with pre-loaded Gold data benefits from the Warehouse query engine and Power BI Direct Lake on Warehouse which has different performance characteristics from Lakehouse Direct Lake.

THE FIX

After writing Gold tables to the Lakehouse, always run `OPTIMIZE` with V-Order to compact files and apply the Fabric-specific read optimization. If SQL endpoint connections are used for Power BI, consider whether loading Gold data into a Warehouse provides better query performance for the report patterns in use. For Direct Lake connections, verify V-Order is applied using `DESCRIBE DETAIL` and confirm file sizes are appropriate for the VertiPaq engine's column segment loading.

05

STORAGE LAYER DESIGN

You stored raw, Silver, and Gold all in the same Lakehouse. Governance and access control became impossible.

THE SITUATION

To keep the architecture simple, all three Medallion layers were built as folder structures within a single Lakehouse. The platform works technically and data flows correctly between layers. But as the team grew, access control became a problem. Teams needing Gold access can also see and accidentally modify Bronze. Data governance cannot be enforced at the layer level.

WRONG THINKING

Engineers assume folder-level organization within a single Lakehouse is sufficient for governance. Since the folders are clearly named and team members are trusted, the lack of hard access boundaries feels acceptable. The governance problem only becomes visible when the team grows or when an incident occurs.

WHAT IS ACTUALLY HAPPENING

Fabric Lakehouse permissions are applied at the Lakehouse level, not at the folder or table level within a Lakehouse. A user with read access to the Lakehouse can read all tables in all folders unless row-level or column-level security is explicitly configured per table. There is no native folder-level permission boundary within a single Lakehouse. Storing all Medallion layers in one Lakehouse means any access grant to that Lakehouse exposes all layers. Bronze data containing PII or raw unvalidated records becomes accessible to anyone who needs Gold access.

THE FIX

Separate Medallion layers into distinct Lakehouses when access control boundaries are required between layers. A Bronze Lakehouse accessible only to the ingestion team, a Silver Lakehouse accessible to data engineers, and a Gold Lakehouse accessible to consumers provides clean permission boundaries. Use OneLake shortcuts to allow Silver pipelines to read from Bronze and Gold pipelines to read from Silver without granting full cross-Lakehouse access. This trades some architectural simplicity for genuine access control that can be enforced without per-table security configurations.

06

STORAGE LAYER DESIGN

You chose Warehouse over Lakehouse for your entire platform. You cannot handle semi-structured data anymore.

THE SITUATION

An architectural decision was made to standardize the entire Fabric platform on the Warehouse for consistency and SQL governance. All ingestion pipelines load data directly into Warehouse tables. After several months, the team needs to ingest JSON logs, nested API responses, and variable-schema event data. The Warehouse cannot handle these ingestion patterns efficiently.

WRONG THINKING

Engineers assume that since the Warehouse is the more mature SQL layer, it should be the right choice for everything. The flexibility trade-off of choosing Warehouse over Lakehouse is not considered during the initial architectural decision. Semi-structured data requirements emerge later when the platform scope expands.

WHAT IS ACTUALLY HAPPENING

Fabric Warehouse is a structured SQL engine optimized for well-defined relational schemas. It does not natively support semi-structured data formats like JSON documents, nested structures, or schema-on-read patterns. Ingesting variable-schema data into Warehouse tables requires upfront schema definition and type mapping. Spark-based ingestion into Delta tables in a Lakehouse handles semi-structured data naturally through schema inference, nested type support, and schema evolution. A platform built entirely on Warehouse cannot adapt to semi-structured data sources without significant rework.

THE FIX

Adopt a hybrid architecture that uses Lakehouse for ingestion and transformation of all data types including semi-structured, and uses Warehouse selectively for structured SQL serving workloads that benefit from the Warehouse query engine. The Lakehouse handles the flexibility of the ingestion layer. The Warehouse handles the performance requirements of the serving layer. Data flows from Lakehouse to Warehouse for the subset of structured data that requires SQL optimization. This layered approach handles both structured and semi-structured sources without architectural compromise.

07

STORAGE LAYER DESIGN

Your team uses both Lakehouse and Warehouse. The same data exists in both places and nobody knows which is authoritative.

THE SITUATION

A Fabric platform evolved organically with some teams using Lakehouses and others using Warehouses. Over time, certain datasets were loaded into both because different teams preferred different access methods. Now the same table exists in the Lakehouse and the Warehouse and the values differ. Reports from different teams contradict each other and nobody can identify which version is correct.

WRONG THINKING

Engineers assume that having data in both places simply provides redundancy and flexibility. The synchronization and ownership problem is recognized but deprioritized because both sources appear to work. The divergence is only discovered when downstream reports produce inconsistent numbers in a business review.

WHAT IS ACTUALLY HAPPENING

When the same data exists in two places without a defined synchronization process and ownership model, divergence is inevitable. Pipelines may load to one source at a different time than the other. Transformation logic may differ between the two versions. Schema changes may be applied to one but not the other. Without a single authoritative source, every consumer must independently verify which version to trust. This is not a technical problem that can be solved with better tooling. It is an architectural governance failure that produces ambiguity at the data level.

THE FIX

Establish a single authoritative source for every dataset. Define whether the Lakehouse or the Warehouse is the source of truth for each domain and document it. If both access paths are needed, use a single authoritative source with a downstream sync: the Lakehouse holds the authoritative Delta table and a pipeline loads the Warehouse from it, or the Warehouse is authoritative and a shortcut or export provides Lakehouse access. Eliminate any pipeline that writes to both simultaneously without a defined ownership model.

08

STORAGE LAYER DESIGN

You exposed the Lakehouse SQL endpoint directly to business users. They are running expensive unoptimized queries against raw tables.

THE SITUATION

Business users were given direct SQL access to the Lakehouse SQL analytics endpoint to enable self-service analytics. Access was granted at the workspace level. Within weeks, several users are running long-running full table scans against Bronze and Silver tables. These queries are consuming significant capacity and slowing down engineering pipelines running in the same workspace.

WRONG THINKING

Engineers assume that read-only SQL access to the Lakehouse is harmless since users cannot modify data. The capacity impact of read queries from business users is not anticipated because the queries look like simple SELECTs. The absence of a governed serving layer is not seen as an architectural gap.

WHAT IS ACTUALLY HAPPENING

Business users with SQL access and limited knowledge of data engineering will naturally query whatever tables are visible to them, including raw and intermediate tables that were never designed for direct consumption. An unfiltered query against a billion-row Bronze table that a business user writes as `SELECT * FROM bronze.events` consumes the same capacity as a production pipeline. The Lakehouse SQL endpoint has no query governor, no cost control per user, and no query timeout enforced at the connection level. Without a governed serving layer, every user with workspace access can trigger expensive queries against any table in the Lakehouse.

THE FIX

Create a dedicated Gold Lakehouse or Warehouse as the only SQL access point for business users. Grant business users access only to the Gold serving layer, not to the Bronze or Silver Lakehouses. Design Gold tables specifically for the query patterns business users need, with appropriate aggregation levels that avoid full scans of raw data. Consider using semantic models as the access layer for non-technical users so query execution is handled by the VertiPaq engine rather than directly against the SQL endpoint.

02

FABRIC ENGINEERING PATTERNS

Chapter 2 OneLake and Shortcuts

Seven scenarios covering how OneLake shortcuts create flexibility that frequently becomes technical debt and where the copy versus shortcut decision determines platform reliability.

09

ONELAKE AND SHORTCUTS

You used shortcuts everywhere to avoid data duplication. Nobody knows where the actual data lives anymore.

THE SITUATION

To minimize storage costs and avoid data duplication, the team created OneLake shortcuts for almost every data sharing requirement. After a year of growth, the platform has dozens of workspaces each containing shortcuts pointing to other workspaces. When a data issue is reported, tracing the data back to its origin requires navigating a chain of shortcuts that nobody fully understands.

WRONG THINKING

Engineers assume that shortcuts are a transparent access mechanism that simply makes data available where needed without adding complexity. The organizational debt of maintaining a large shortcut graph is not anticipated when shortcuts are first introduced as a convenience pattern.

WHAT IS ACTUALLY HAPPENING

OneLake shortcuts are powerful but they create implicit dependencies between workspaces. A shortcut in Workspace B pointing to Workspace A means that any change, deletion, or permission modification in Workspace A immediately affects Workspace B. When shortcuts chain through multiple workspaces, a single change at the origin propagates through every downstream shortcut without warning. Debugging data issues requires understanding the full shortcut dependency graph which grows non-linearly as the platform scales. There is no native lineage view that shows the full chain of shortcut dependencies across workspaces.

THE FIX

Use shortcuts intentionally and document every shortcut with its source workspace, purpose, and owner. Establish a policy that shortcuts are used for read access to authoritative sources, not as a substitute for a defined data sharing architecture. For frequently consumed datasets, consider materializing a copy in the consuming workspace rather than maintaining a long shortcut chain. Implement a shortcut registry as a simple metadata table that records every shortcut in the platform, its source, its consumers, and the last validation date. Audit the registry regularly to identify stale or undocumented shortcuts.

10

ONELAKE AND SHORTCUTS

You copied data into every workspace instead of using shortcuts. Storage costs tripled.

THE SITUATION

To avoid the dependency complexity of shortcuts, the team adopted a policy of copying data into every workspace that needs it. Each team maintains their own copy of shared reference data. Over time, the same datasets exist in five or six workspaces. Storage costs have grown significantly and pipeline maintenance is consuming a large share of engineering capacity to keep all copies synchronized.

WRONG THINKING

Engineers assume that data independence justifies the storage cost. Having a local copy feels more reliable than depending on another team's workspace. The total storage cost of maintaining six copies of the same data is not calculated until it appears on the capacity bill.

WHAT IS ACTUALLY HAPPENING

Copying data into every workspace that needs it solves the dependency problem but creates a synchronization problem. Each copy must be refreshed on a schedule that matches the source update frequency. If any copy falls behind, the workspace is serving stale data. If the source schema changes, all copies must be updated simultaneously or they diverge. The storage cost compounds with every new workspace and every new dataset that follows the same pattern. What appeared to be independence becomes a maintenance burden that scales with platform growth.

THE FIX

Use a tiered approach: shortcuts for authoritative shared reference data that changes infrequently and is owned by a single team, and copies for data that must be transformed before consumption or that has strict SLA requirements that cannot tolerate source dependency. Define clear ownership for every shared dataset and publish it through a designated authoritative workspace that all consumers shortcut to. This gives consumers read access to authoritative data without duplicating storage, while reserving copies for cases where transformation or SLA independence genuinely justifies the cost.

11

ONELAKE AND SHORTCUTS

Your shortcut points to external ADLS. Queries through it are 10x slower than native OneLake tables.

THE SITUATION

A OneLake shortcut was created to an Azure Data Lake Storage Gen2 account to avoid migrating data into Fabric immediately. The shortcut appears correctly in the Lakehouse and queries return correct results. But queries through the shortcut consistently take 10 times longer than equivalent queries on native Lakehouse Delta tables. The data volumes are comparable.

WRONG THINKING

Engineers assume that OneLake shortcuts are a transparent layer that provides equivalent performance regardless of the underlying storage location. The shortcut is treated as equivalent to a native table because it looks identical in the Lakehouse explorer.

WHAT IS ACTUALLY HAPPENING

OneLake shortcuts to external storage do not benefit from the same optimizations as native OneLake Delta tables. Native OneLake tables benefit from V-Order, local file caching, and metadata co-location that reduces planning overhead. External ADLS shortcuts require the Fabric runtime to make network calls to the external storage account for every file metadata request and data read. There is no local caching equivalent for external shortcuts. The performance gap reflects the difference between reading from optimized local storage versus making repeated network calls to external storage across a storage account boundary.

THE FIX

Migrate data from external ADLS into native Lakehouse Delta tables using a pipeline when query performance is a requirement. After migration, run `OPTIMIZE` to apply V-Order and compact files. Keep the external ADLS shortcut only for initial ingestion workflows where data is read once and transformed into native Delta tables. Never use external storage shortcuts as the primary serving layer for analytical queries. The shortcut is a migration convenience, not a performance-equivalent substitute for native storage.

12

ONELAKE AND SHORTCUTS

You deleted a table that multiple shortcuts were pointing to. Downstream workspaces broke silently.

THE SITUATION

A Lakehouse table that had been superseded by a new version was deleted as part of a cleanup exercise. The engineer deleting the table checked the workspace for any dependencies and found none. Within hours, three other teams reported broken pipelines and failed reports. Investigation revealed that all three teams had created shortcuts pointing to the deleted table in their own workspaces.

WRONG THINKING

Engineers assume that searching their own workspace for references to a table is sufficient before deleting it. Cross-workspace shortcut dependencies are not visible within the source workspace and are not checked. The deletion is treated as a local operation with local impact.

WHAT IS ACTUALLY HAPPENING

OneLake shortcuts are one-directional references. The source workspace has no visibility into which other workspaces have created shortcuts pointing to its tables. Deleting a table in the source workspace does not trigger any notification to consuming workspaces. All shortcuts pointing to the deleted table immediately start failing without any warning to the source team or the consuming teams. The impact is only discovered when downstream workloads next attempt to access the shortcut.

THE FIX

Before deleting any table that other teams might be consuming, broadcast a deprecation notice with a defined sunset date rather than deleting immediately. Use the Fabric workspace lineage view to check for any registered dependencies within the workspace, but accept that cross-workspace shortcut dependencies cannot be detected from the source side. Establish a data catalog or shortcut registry that teams update when they create shortcuts to external workspaces so the source team can check the registry before deleting tables. Allow a minimum notice period between announcing deprecation and completing deletion.

13

ONELAKE AND SHORTCUTS

You created a shortcut to share data between teams. The source team changed the schema and the consuming team found out from broken reports.

THE SITUATION

Team A owns a Gold table and created a shortcut so Team B could access it without data duplication. Both teams agreed this was a clean architecture. Three months later, Team A added columns, renamed an existing column, and changed a data type as part of a routine schema update. Team B's pipelines and Power BI reports broke immediately. Team A was not aware Team B had built production workloads on the shortcut.

WRONG THINKING

Team A assumes their Gold table is their domain and schema changes are their decision. Creating a shortcut feels like a read-only sharing arrangement where the consuming team takes responsibility for keeping up with changes. The coupling that the shortcut creates between the two teams' release cycles is not discussed when the shortcut is established.

WHAT IS ACTUALLY HAPPENING

A shortcut to another team's table creates a hard dependency on that table's schema. Unlike an API with a versioned contract, a OneLake shortcut exposes the raw table structure with no versioning, no backward compatibility guarantee, and no change notification mechanism. The consuming team's workloads are tightly coupled to the source team's schema. Any schema change by the source team is immediately a breaking change for every consumer of the shortcut. This is not a technical limitation but an architectural coupling problem that shortcuts make invisible.

THE FIX

Treat shared data exposed through shortcuts as a data contract with formal versioning. Before creating a shortcut arrangement between teams, define what schema changes require advance notice, what the notice period is, and who is responsible for communicating changes. For tables that require stability guarantees, consider exposing a dedicated stable view or materialized copy rather than a shortcut directly to the working table. Establish a change management process for tables that have external shortcuts, similar to how an API breaking change is managed.

14

ONELAKE AND SHORTCUTS

Your OneLake has data scattered across dozens of workspaces with no consistent folder structure. Finding anything requires tribal knowledge.

THE SITUATION

Over two years, the Fabric platform grew organically. New workspaces were created as needed, Lakehouses were named by whoever created them, and tables were organized however seemed logical at the time. New engineers joining the team spend days figuring out where data lives. Data lineage investigations require asking specific people because no documentation reflects the current state.

WRONG THINKING

Engineers assume that as long as data engineers know where everything is, organizational structure does not matter. Documentation is considered a secondary concern that will be addressed later. The tribal knowledge problem only becomes visible when team members leave or when the platform reaches a scale where no single person can hold the full picture.

WHAT IS ACTUALLY HAPPENING

An ungoverned OneLake structure accumulates entropy over time. Each engineer makes locally reasonable naming and organization decisions that collectively produce a platform that is difficult to navigate. Without a consistent workspace naming convention, Lakehouse naming standard, and table organization pattern, the cognitive overhead of understanding the platform grows with every new addition. The cost of this entropy is paid continuously by every engineer who needs to find, validate, or trace data, and it compounds as the platform grows.

THE FIX

Establish and enforce a naming convention for workspaces, Lakehouses, and tables before the platform grows beyond the initial team. Define a standard folder structure within each Lakehouse based on Medallion layers or data domains. Document the convention and enforce it through code review or automated naming validation in deployment pipelines. For existing ungoverned platforms, run a naming audit and migration sprint. The cost of establishing order early is a fraction of the cost of navigating disorder at scale.

15

ONELAKE AND SHORTCUTS

You gave a team OneLake read access to raw data. They built reports directly on unvalidated Bronze layer data.

THE SITUATION

To accelerate a business team's self-service analytics, OneLake read access was granted to a Lakehouse containing Bronze layer data. The intention was that the team would use this for ad-hoc exploration only. Six months later, the team has built production Power BI reports directly on Bronze tables. These reports are being used in executive dashboards and the business team considers them authoritative.

WRONG THINKING

Engineers assume that granting read-only access to raw data is safe because users cannot modify it. The risk is not in modification but in consumption. Business users building production reports on Bronze data is not anticipated because the instruction to use it only for exploration was informal and not enforced architecturally.

WHAT IS ACTUALLY HAPPENING

Raw Bronze data is unvalidated, inconsistently formatted, and subject to change as source systems evolve. Schema changes in the source system flow directly into Bronze tables and immediately break any reports built on them. Data quality issues that are caught and corrected in the Silver layer appear unfiltered in Bronze reports. When business teams build on Bronze and treat it as authoritative, they are consuming data that the engineering team never intended to guarantee. The informal instruction to use Bronze only for exploration has no enforcement mechanism and will be forgotten or overridden by business pressure.

THE FIX

Remove direct business user access to Bronze and Silver Lakehouses. Provide access only to Gold serving layers that have defined quality guarantees, stable schemas, and ownership commitments. If exploratory access to intermediate data is required, create a dedicated exploration environment that is clearly labeled as non-production and has no connection to executive reporting. The architecture should make it physically impossible to accidentally build production reports on unvalidated data by not granting access to the raw layers.

03

FABRIC ENGINEERING PATTERNS

Chapter 3 Medallion Architecture

Eight scenarios covering Medallion architecture anti-patterns that only become visible after the platform has been running for months and data volumes and team sizes have grown.

16

MEDALLION ARCHITECTURE

You built Bronze, Silver, and Gold as three separate Lakehouses. Cross-layer queries require shortcuts and are slow.

THE SITUATION

A Fabric platform was designed with one Lakehouse per Medallion layer to achieve clean separation. Bronze, Silver, and Gold each live in separate Lakehouses in separate workspaces. Pipelines move data between layers correctly. But data engineers need to join Bronze and Silver data during debugging and troubleshooting, and cross-Lakehouse queries through shortcuts are slow and cumbersome.

WRONG THINKING

Engineers assumed that physical separation of Medallion layers is always the right architecture because it provides the cleanest boundaries. The operational cost of cross-layer access during debugging, incident investigation, and development is not considered when the architecture is designed.

WHAT IS ACTUALLY HAPPENING

Three separate Lakehouses for three Medallion layers is the right choice when strict access control boundaries between layers are required. When the team is small and access control is not a primary concern, three separate Lakehouses adds operational complexity without proportional benefit. Cross-Lakehouse queries require shortcuts, which add a network hop. Development workflows that need to inspect data at multiple layers simultaneously become cumbersome. The architecture solved a governance problem the team did not yet have while creating an operational problem they face daily.

THE FIX

Match the Medallion architecture to the actual access control requirements of the team. For small teams with no strict layer access boundaries, a single Lakehouse with separate schema namespaces or folder structures per layer is simpler and more efficient. Introduce separate Lakehouses per layer only when the team size, data sensitivity, or regulatory requirements genuinely require access control between layers. Do not apply enterprise-scale architecture patterns to platforms that do not yet have enterprise-scale governance requirements.

17

MEDALLION ARCHITECTURE

You built Medallion with a single Lakehouse. All three layers share the same permissions and teams can access Bronze directly.

THE SITUATION

To keep the architecture simple, all three Medallion layers were implemented as table namespaces within a single Lakehouse. The platform works and pipelines are efficient. But as the team grew and external analysts were onboarded, access control became impossible to enforce. Analysts with Gold access can also see and query Bronze tables containing raw PII data. There is no way to restrict access to specific tables within the Lakehouse without implementing row and column level security on every sensitive table individually.

WRONG THINKING

Engineers chose a single Lakehouse for simplicity and assumed that team discipline would prevent accidental access to lower layers. The access control requirement was deprioritized during initial design. The problem only became visible when the team grew beyond the founding engineers who understood the informal conventions.

WHAT IS ACTUALLY HAPPENING

A single Lakehouse grants Lakehouse-level access to all tables within it. There is no native folder or schema-level permission boundary within a Fabric Lakehouse. Granting read access to the Gold tables implicitly grants read access to every table in the Lakehouse including Bronze tables with raw unvalidated data. Implementing per-table security as a retrofit requires configuring row-level security or column-level security on every sensitive table individually, which is a significant ongoing maintenance burden as new tables are added.

THE FIX

When access control between Medallion layers is a requirement, separate the layers into distinct Lakehouses from the start. The cost of separating later after pipelines and reports have been built is significantly higher than designing the separation upfront. Use OneLake shortcuts to provide Silver pipelines with read access to Bronze and Gold pipelines with read access to Silver, while keeping the workspace and Lakehouse permissions separate per layer. Accept the slight operational overhead of cross-Lakehouse references in exchange for enforceable access boundaries.

18

MEDALLION ARCHITECTURE

Your Gold layer is slower to query than Silver. You built Gold but never optimized it.

THE SITUATION

A Medallion architecture was implemented correctly with Bronze, Silver, and Gold layers. Gold tables contain pre-aggregated and denormalized data specifically designed for reporting. Despite containing less data than Silver, Gold queries through Power BI or the SQL endpoint are slower than equivalent queries on the Silver tables they were built from.

WRONG THINKING

Engineers assume that denormalization and pre-aggregation automatically produce faster queries because the data volume is smaller. The Gold tables are populated by pipelines but no post-write optimization is applied. The assumption is that writing the right data is sufficient.

WHAT IS ACTUALLY HAPPENING

Gold tables are only faster than Silver if they are also optimized for the read patterns they serve. Delta tables written by Spark pipelines without OPTIMIZE applied have many small Parquet files, poor columnstore compression, and no V-Order applied. A Gold table with 10 million rows spread across 500 small files will query slower than a Silver table with 100 million rows in 50 large optimized files. The data volume advantage of Gold is negated by the file structure disadvantage. Writing correct data to Gold is the pipeline's job. Making Gold queryable efficiently requires post-write maintenance.

THE FIX

Add an OPTIMIZE step with V-Order to every pipeline that writes to Gold tables. Run `OPTIMIZE tablename VORDER` after every Gold write as a mandatory pipeline step before the downstream semantic model refresh or consumer notification. For Gold tables with date partitions, run OPTIMIZE only on the partitions that were updated rather than the full table to reduce maintenance time. Verify optimization was applied using `DESCRIBE DETAIL tablename` and confirm file counts are appropriate for the table size. A Gold table with 10 million rows should have no more than a handful of Parquet files after optimization.

19

MEDALLION ARCHITECTURE

Your Bronze layer is being used directly by downstream teams. You have no control over data quality in reports.

THE SITUATION

The Bronze Lakehouse was made accessible to several downstream teams for convenience during an early project phase. The plan was to eventually migrate consumers to the Gold layer once it was built. Six months later, Gold is built but downstream teams have not migrated. Their pipelines and reports are built directly on Bronze tables and any attempt to change or deprecate Bronze tables causes immediate breakages.

WRONG THINKING

Engineers assumed that downstream consumers would migrate to Gold once it was available because Gold is clearly better data. The migration was treated as the consuming team's responsibility. Without enforcement, migration never happened because the consuming teams had no incentive to do the work when Bronze was already working for them.

WHAT IS ACTUALLY HAPPENING

Once a data layer is consumed in production, it becomes load-bearing regardless of its intended purpose. Bronze tables that were supposed to be internal become de facto production serving tables the moment downstream pipelines depend on them. The engineering team loses the ability to evolve Bronze schemas, implement quality fixes, or restructure the ingestion layer without coordinating with every downstream consumer. The Medallion architecture's promise of clean layer separation breaks down as soon as the internal layers are exposed.

THE FIX

Never grant production access to Bronze or Silver layers to teams outside the data engineering team. From day one, Gold should be the only layer with external consumer access. If a Gold layer does not yet exist, deliver a minimal Gold layer before onboarding any downstream consumers rather than granting temporary Bronze access that will become permanent. Enforce this architecturally by keeping Bronze and Silver in workspaces with no external access grants rather than relying on team discipline or future migration promises.

20

MEDALLION ARCHITECTURE

You reprocess the entire Bronze to Silver pipeline every day. It takes 6 hours even when only 1 percent of data changed.

THE SITUATION

A daily pipeline reads all Bronze data, applies transformations, and writes the full output to Silver. The pipeline runs correctly and Silver always reflects the current state. But as Bronze data volumes grew, the full reprocess pipeline now takes 6 hours, consuming significant capacity and delaying downstream Gold refreshes and report updates.

WRONG THINKING

Engineers built the full reprocess pattern initially because it was simple and guaranteed correctness. Incremental processing was deprioritized as a future optimization. The 6-hour runtime was acceptable at smaller data volumes and the problem crept up gradually as data accumulated over months.

WHAT IS ACTUALLY HAPPENING

A full reprocess pipeline reads, transforms, and writes every row on every run regardless of whether it changed. At small data volumes this is a reasonable simplicity trade-off. As data volumes grow, the fixed cost per run grows linearly with the total data size rather than with the daily change volume. A platform where 99 percent of data is unchanged should not pay the cost of processing 100 percent of data every day. The architecture was not designed for the data volume it now operates at.

THE FIX

Implement incremental processing using Delta Lake change data capture. Use `readStream` with Delta as a streaming source to process only new or changed rows since the last successful run, tracked via checkpointing. For batch pipelines, use watermark columns or Delta table version tracking to identify changed records. Store the last processed watermark in a control table and use it to filter the Bronze source on each run. Test incremental logic thoroughly against edge cases including late-arriving data, source corrections, and pipeline failures before replacing the full reprocess pattern.

21

MEDALLION ARCHITECTURE

Your Silver layer has 200 tables. Nobody agrees on what Silver means and data quality is inconsistent.

THE SITUATION

The Silver layer grew organically as different teams added tables for their specific use cases. After two years, Silver contains 200 tables from a dozen different pipelines written by different engineers. Some Silver tables have strict data quality checks. Others are barely different from Bronze. Some are denormalized for specific consumers. There is no consistent definition of what Silver means on this platform.

WRONG THINKING

Engineers treated Silver as a general-purpose intermediate layer where any partially transformed data could live. Each team defined their own standards for their own Silver tables. The lack of a platform-wide Silver definition was not seen as a problem until consumers started discovering inconsistent data quality across Silver tables.

WHAT IS ACTUALLY HAPPENING

When Silver has no consistent definition, it provides no consistent guarantee. A consumer querying a Silver table cannot know whether it has been deduplicated, validated, standardized, or simply renamed from Bronze. The Silver label becomes meaningless. Downstream consumers must validate data quality independently for every Silver table they consume, eliminating the core value that a properly defined Silver layer should provide. A Silver layer with 200 tables and no standards is operationally equivalent to a Bronze layer with extra steps.

THE FIX

Define a Silver contract for the platform that specifies exactly what every Silver table must guarantee: deduplication, null handling for key columns, data type standardization, and validation rule coverage. Enforce the contract through a shared data quality framework that every Silver pipeline must pass before writing. Tables that do not meet the Silver contract should either be upgraded to meet it or moved to a separate exploratory layer that does not carry the Silver quality implication. Reduce the Silver table count by identifying redundant tables serving overlapping use cases.

22

MEDALLION ARCHITECTURE

You built Gold tables for every possible use case. Maintenance cost is higher than the value delivered.

THE SITUATION

In an effort to serve every business team efficiently, the data engineering team built a dedicated Gold table for every reporting use case. After 18 months there are 150 Gold tables, each slightly different from others. Maintaining 150 tables, keeping them current, and handling schema changes across all of them consumes more engineering time than building new capabilities. Business teams still request new Gold tables faster than old ones are deprecated.

WRONG THINKING

Engineers treated Gold table creation as the primary value delivery mechanism. Every new business request was answered with a new Gold table. The total maintenance burden of the Gold layer was not tracked as a cost because each individual table was small and fast to maintain. The aggregate burden only became visible when the team ran out of capacity for new work.

WHAT IS ACTUALLY HAPPENING

A Gold layer that grows without a deprecation process accumulates technical debt at the same rate as the business generates new requests. Each new Gold table adds ongoing maintenance cost: pipeline runs, storage, monitoring, schema evolution, and consumer communication. Without measuring and managing this cost, the Gold layer eventually consumes all available engineering capacity with maintenance and leaves nothing for new development. More Gold tables is not always better. The right Gold layer is the smallest one that covers the actual business requirements.

THE FIX

Implement a Gold table lifecycle policy. Every Gold table must have a declared owner, a consumer list, and a review date. Tables with no active consumers are deprecated on a defined schedule. Before creating a new Gold table, check whether an existing table can be extended rather than creating a parallel one. Measure the total maintenance cost of the Gold layer quarterly and treat it as a budget that constrains new Gold table creation. Prioritize reusable Gold tables that serve multiple consumers over single-use tables built for one specific report.

23

MEDALLION ARCHITECTURE

Your Medallion layers have circular dependencies. A failure in Gold triggers a rerun of Silver which breaks Gold again.

THE SITUATION

A Fabric platform has pipelines that move data from Bronze to Silver to Gold. Over time, additional pipelines were added to handle special cases: a Gold table that feeds back into Silver for enrichment, a Silver table that reads from a Gold lookup table. After a Gold pipeline failure, the incident response triggered reruns that created circular dependency loops. Pipelines ran out of order and produced inconsistent data across layers.

WRONG THINKING

Engineers added back-references from Gold to Silver because it solved a specific business requirement efficiently at the time. The architectural implication of introducing a cycle into what should be a directed acyclic dependency graph was not considered. Each back-reference seemed like a small exception to the rule.

WHAT IS ACTUALLY HAPPENING

Medallion architecture depends on a strictly directed data flow: Bronze feeds Silver feeds Gold. Any reference that flows in the reverse direction creates a cycle. In a cyclic dependency graph, a failure in any node can propagate in both directions through the cycle, making it impossible to determine a safe rerun order. Pipeline orchestration tools cannot automatically handle cycles and manual intervention is required for every incident. The more back-references exist, the more fragile the recovery process becomes and the harder it is to reason about data freshness and consistency.

THE FIX

Audit all pipelines and identify any data flow that moves from a higher Medallion layer to a lower one. Eliminate every back-reference by restructuring the data flow. If Gold data is needed to enrich Silver, the enrichment source belongs in Silver or in a dedicated reference layer, not in Gold. If a lookup table currently in Gold is needed by Silver pipelines, move it to Silver or to a shared reference Lakehouse that sits outside the Medallion flow. Enforce the no-back-reference rule as a pipeline architecture standard and review it in every pipeline design review.

04

FABRIC ENGINEERING PATTERNS

Chapter 4 Workspace and Capacity

Seven scenarios covering how workspace topology and capacity design decisions create contention, governance failures, and cost surprises that cannot be easily fixed once the platform is built.

24

WORKSPACE AND CAPACITY

You put everything in one workspace. One team's heavy workload is slowing down everyone else.

THE SITUATION

All Fabric workloads were placed in a single workspace to simplify governance and management. Engineering pipelines, data science notebooks, Power BI reports, and the Warehouse all share the same workspace and the same Fabric capacity. When a data engineer runs a heavy Spark notebook or a large pipeline batch, report load times for business users increase and Power BI refresh jobs queue.

WRONG THINKING

Engineers assumed that a single workspace is simpler to manage and that Fabric capacity handles resource sharing automatically. The impact of heavy engineering workloads on interactive business user workloads was not anticipated because each workload type was tested in isolation.

WHAT IS ACTUALLY HAPPENING

Fabric capacity is shared across all items in workspaces assigned to that capacity. A Spark notebook consuming 80 percent of available capacity leaves only 20 percent for concurrent Power BI queries and pipeline runs. Fabric has some automatic throttling and queuing behavior but it does not prevent one workload type from significantly impacting others when capacity is constrained. Placing all workload types in one workspace assigned to one capacity creates resource contention that cannot be resolved without architectural changes.

THE FIX

Separate workloads by type across multiple workspaces assigned to appropriate capacity tiers or separate capacities. Assign heavy batch engineering workloads to a workspace on a capacity sized for compute-intensive Spark jobs. Assign interactive reporting workloads to a separate workspace on a capacity optimized for low-latency queries. This separation allows each workload type to consume capacity without impacting others and allows capacity sizing decisions to be made independently per workload type.

25

WORKSPACE AND CAPACITY

You created a separate workspace per team. Sharing data between teams requires complex shortcut management.

THE SITUATION

To give each team autonomy and prevent resource contention, a separate workspace was created for every team. After a year, the platform has 15 workspaces. Data sharing between teams requires creating and maintaining shortcuts across workspace boundaries. Every cross-team data dependency requires a shortcut agreement, permission grants, and ongoing maintenance. Engineers spend significant time managing the shortcut graph rather than building pipelines.

WRONG THINKING

Engineers assumed that one workspace per team is the right default for isolation and autonomy. The coordination cost of cross-team data sharing was not anticipated when the workspace topology was designed. Each individual workspace felt clean and simple. The aggregate complexity of 15 workspaces sharing data became visible only after the graph grew.

WHAT IS ACTUALLY HAPPENING

One workspace per team optimizes for team autonomy at the expense of data sharing efficiency. As the number of teams grows, the number of cross-workspace data dependencies grows non-linearly. Each dependency requires shortcut creation, permission management, and change coordination. With 15 workspaces and multiple cross-team dependencies each, the shortcut graph becomes a significant operational overhead. The autonomy benefit of workspace isolation is real but it has a coordination cost that must be managed explicitly.

THE FIX

Design workspace topology around data domains rather than organizational teams. Group Lakehouses and pipelines that serve the same data domain into a shared domain workspace regardless of which team owns them. Teams that consume data from multiple domains access a shared serving workspace rather than building cross-team shortcut dependencies. Reserve separate workspaces for workloads that genuinely require isolation such as production versus development environments or regulated data that requires strict access control. Aim for the minimum number of workspaces that satisfies the actual access control and isolation requirements.

26

WORKSPACE AND CAPACITY

You built dev and prod in the same workspace. A developer accidentally overwrote a production table.

THE SITUATION

To avoid the complexity of managing multiple workspaces, development and production workloads were placed in the same workspace. Pipelines were prefixed with dev_ or prod_ to distinguish them. One afternoon, a developer testing a pipeline used the wrong table name and ran a write operation that truncated a production Gold table that downstream reports depend on. Recovery required restoring from a Delta table time travel snapshot.

WRONG THINKING

Engineers assumed that naming conventions and team discipline were sufficient to prevent accidental cross-environment operations. The workspace was treated as a single logical environment with internal naming to separate dev from prod. The physical separation that a separate workspace provides was considered unnecessary overhead.

WHAT IS ACTUALLY HAPPENING

Naming conventions are not access controls. In a shared workspace, any pipeline with write access to the workspace can write to any table regardless of the table's intended environment. There is no technical enforcement of the dev/prod boundary when both environments share the same workspace and the same permissions. Discipline-based separation works until it does not and when it fails in production, the impact is immediate and visible to business users.

THE FIX

Use separate workspaces for development and production environments. This is the minimum viable environment separation for any platform serving business users. Separate workspaces enforce the dev/prod boundary at the permission level rather than relying on naming conventions. Use Fabric deployment pipelines to promote tested artifacts from dev to prod in a controlled and auditable way. The overhead of managing two workspaces is a small and fixed cost. The overhead of recovering from a production incident caused by a missing boundary is large and unpredictable.

27

WORKSPACE AND CAPACITY

You sized Fabric capacity for peak load. You are paying for idle compute during off-peak hours.

THE SITUATION

Fabric capacity was provisioned at a tier large enough to handle the peak concurrent workload: morning ETL runs combined with business user report access. The capacity handles peak hours well. But for 16 hours a day when pipelines are not running and users are not active, the platform is running at 5 percent utilization while paying for 100 percent of the capacity cost.

WRONG THINKING

Engineers assumed that right-sizing capacity means choosing the tier that handles peak load. The cost of paying for idle capacity during off-peak hours was not calculated as part of the capacity decision. The capacity tier was chosen to eliminate performance problems, not to optimize total cost.

WHAT IS ACTUALLY HAPPENING

Fabric capacity is billed continuously regardless of utilization. A capacity tier chosen for peak load pays the full cost of that tier 24 hours a day, 7 days a week. If peak load occurs for 8 hours a day and the remaining 16 hours are idle, the effective utilization is 33 percent but the cost is 100 percent of the peak tier price. Fabric capacity does support pausing and resuming, but pausing a capacity that has active workloads or scheduled pipelines requires careful orchestration. The mismatch between peak sizing and average utilization is a significant cost inefficiency for platforms with variable workload patterns.

THE FIX

Evaluate whether Fabric capacity pause and resume can be implemented for off-peak hours. Pause the capacity after the last scheduled pipeline completes each night and resume it before the first scheduled pipeline starts each morning. Automate the pause and resume through Azure Automation or a Logic App triggered by a schedule. For workloads with more variable patterns, consider whether multiple smaller capacities assigned to different workspace groups allows more granular scaling than a single large capacity. Monitor actual capacity unit consumption using the Fabric Monitoring Hub to identify the true peak versus average utilization ratio before making sizing decisions.

28

WORKSPACE AND CAPACITY

You assigned all workloads to one Fabric capacity. A runaway pipeline is consuming all capacity and blocking reports.

THE SITUATION

All workspaces are assigned to a single Fabric capacity. A pipeline with a misconfigured parallelism setting or an unoptimized Spark job occasionally consumes the majority of available capacity units. When this happens, Power BI report queries queue, interactive notebook sessions slow dramatically, and other pipelines are delayed. The incident resolves when the runaway workload completes but it recurs unpredictably.

WRONG THINKING

Engineers assumed that a single capacity assignment is simpler to manage and that Fabric's built-in workload management handles resource sharing fairly. The risk of one heavy workload starving all other workloads on the same capacity was not considered as an architectural concern.

WHAT IS ACTUALLY HAPPENING

Fabric capacity has a finite pool of capacity units that are shared across all workspaces assigned to it. A single runaway workload that requests a large number of capacity units can consume a disproportionate share of the pool, leaving insufficient units for other concurrent workloads. Fabric does have smoothing and throttling mechanisms but these operate over time windows and do not prevent short-term resource starvation. Assigning all workloads to one capacity creates a single point of resource contention with no isolation between workload types.

THE FIX

Assign different workload types to separate Fabric capacities where the business impact of resource contention justifies the cost of separate capacities. At minimum, separate interactive reporting workloads from batch engineering workloads onto different capacities so a runaway batch pipeline cannot impact report availability. For batch workloads, implement pipeline activity limits and Spark session configuration caps to prevent individual jobs from consuming disproportionate capacity. Use the Fabric Monitoring Hub to identify the top capacity consumers and set reasonable limits before they become incidents.

29

WORKSPACE AND CAPACITY

Your workspace has no naming convention for Fabric items. Finding anything requires searching manually.

THE SITUATION

A Fabric workspace grew organically over 18 months. Pipelines, notebooks, Lakehouses, and semantic models were named by whoever created them at the time. Some items are named by function, some by the creating engineer's initials, some by date, and some by a project code that is no longer used. New engineers joining the team spend days finding the right items. When an incident occurs, identifying the relevant pipeline takes longer than fixing the actual problem.

WRONG THINKING

Engineers assumed that search functionality in the Fabric workspace would compensate for inconsistent naming. Each item was named in a way that made sense to the engineer who created it at the time. The aggregate confusion of inconsistent naming only becomes visible when the workspace grows beyond what a single person can hold in memory.

WHAT IS ACTUALLY HAPPENING

A workspace without naming conventions is a workspace that only its creators can navigate efficiently. Search helps find items by name but only when you know what to search for. Incident response, impact analysis, and onboarding all depend on being able to identify items by their purpose from their name alone. Inconsistent naming forces every engineer to maintain a mental map of the workspace that is never fully accurate and degrades as people leave and join the team.

THE FIX

Establish a naming convention that encodes environment, layer, domain, and purpose into every item name. A pattern like `env_layer_domain_itemtype_description` gives every item a self-describing name. For example: `prod_gold_sales_pipeline_daily_aggregation`. Apply the convention retroactively to all existing items in a naming sprint before the workspace grows further. Enforce the convention in code review and deployment pipeline validation. The cost of consistent naming is minutes per item. The cost of inconsistent naming is hours per incident.

30

WORKSPACE AND CAPACITY

You promoted your pipeline directly from dev to prod workspace. The pipeline is pointing at dev data sources in production.

THE SITUATION

A pipeline was developed and tested in the dev workspace. When ready, it was manually recreated in the prod workspace by copying the pipeline JSON and pasting it into the Fabric UI. The pipeline runs in production without errors. But after a week, it is discovered that several activity configurations still reference dev Lakehouses and dev connection strings. The pipeline has been writing data to the dev environment from the prod workspace.

WRONG THINKING

Engineers assumed that manually recreating a pipeline in a new workspace was a reliable promotion method. The pipeline was tested and it ran without errors so the output was assumed to be correct. The connection references inside individual activities were not systematically reviewed after recreation.

WHAT IS ACTUALLY HAPPENING

Manual pipeline recreation by copy-paste is an error-prone promotion method because it relies on the engineer to correctly identify and update every environment-specific reference in every activity. A pipeline with 15 activities may have environment-specific references in 10 of them. Missing even one means production workloads silently operate against development resources. The absence of errors during execution does not confirm that the pipeline is using the correct environment because dev and prod data sources are both accessible and both return data without errors.

THE FIX

Use Fabric deployment pipelines for all promotions from dev to prod. Configure deployment pipeline parameters to automatically remap data source connections from dev to prod during promotion. After every promotion, run a post-deployment validation script that queries the pipeline definition and confirms all connection references point to production resources. Never use manual copy-paste as a promotion method for production pipelines. If deployment pipelines are not yet configured, treat the connection audit as a mandatory promotion checklist item with sign-off required before the pipeline runs in production.

05

FABRIC ENGINEERING PATTERNS

Chapter 5 Security Architecture

Seven scenarios covering where the gap between apparent security and actual security is widest in Fabric and how access control decisions made at design time create vulnerabilities that are hard to close later.

31

SECURITY ARCHITECTURE

You enforced RLS at the Lakehouse, SQL endpoint, and semantic model simultaneously. Nobody knows which layer is doing the work.

THE SITUATION

A data platform enforces row-level security at three layers: OneLake access roles on the Lakehouse, RLS filters on the SQL analytics endpoint, and RLS roles in the Power BI semantic model. All three layers are active. Security audits pass. But when a data access incident is investigated, nobody can determine which layer blocked or allowed the access. Performance is also significantly worse than expected for secured users.

WRONG THINKING

Engineers added security at each layer as different teams requested it, each believing their layer was the definitive enforcement point. Having security at multiple layers felt more secure than relying on a single layer. The operational complexity of debugging multi-layer security and its performance impact were not considered.

WHAT IS ACTUALLY HAPPENING

Overlapping RLS at multiple layers compounds query execution cost. For a secured Power BI user, the query path evaluates the semantic model RLS in the DAX engine, then passes the filtered query to the SQL endpoint which evaluates its own RLS, which then checks OneLake access roles. Each evaluation adds latency. When an access decision produces an unexpected result, debugging requires understanding which layer is making the decision and in what order. With three independent security layers each with their own rule sets, a change in any one layer can produce unexpected interactions with the others.

THE FIX

Define a single authoritative security enforcement layer and remove redundant security from the other layers. For most Fabric architectures, semantic model RLS is the right enforcement point for report consumers because it is closest to the consumption layer and has the least performance overhead. OneLake access roles are appropriate for controlling which services and pipelines can read raw data at the storage level. SQL endpoint RLS should be used only when the SQL endpoint is the primary access path and semantic model RLS is not available. Document which layer enforces security for each access path and audit it regularly.

32

SECURITY ARCHITECTURE

You gave everyone Contributor access to the main workspace. A misconfigured pipeline deleted three months of data.

THE SITUATION

To move quickly during the build phase, all team members were granted Contributor access to the main Fabric workspace. The project delivered on time. Three months later, a developer testing a pipeline deletion script ran it against the wrong environment variable and deleted the primary Gold Lakehouse along with three months of processed data. Delta time travel recovered most of it but the incident took two days and caused significant downstream disruption.

WRONG THINKING

Engineers assumed that all team members being trusted professionals made Contributor access a reasonable default. The risk of accidental rather than malicious data destruction was not considered a significant threat because every team member had legitimate reasons to be in the workspace. Access was never reviewed after the initial project phase.

WHAT IS ACTUALLY HAPPENING

Contributor access in Fabric workspaces grants the ability to create, modify, and delete all items in the workspace including Lakehouses and their data. In a workspace where multiple engineers have Contributor access, a single command error can cause irreversible data loss. The threat model for production data platforms is not primarily malicious actors. It is accidental operations by legitimate users with too much access. The wider the set of users with destructive permissions, the higher the probability that an accident occurs over time.

THE FIX

Apply least-privilege access to all workspace roles. Engineers who need to run pipelines and read data should have Member or Viewer access, not Contributor. Reserve Contributor access for engineers who genuinely need to create or modify workspace items and limit this to a small set. For production workspaces, require a formal access request and review process for Contributor grants. Review workspace access quarterly and revoke Contributor access from anyone who no longer needs it. Use service principals rather than personal accounts for all automated pipeline execution.

33

SECURITY ARCHITECTURE

You secured data at the workspace level only. Users with workspace access can query all tables including sensitive ones.

THE SITUATION

A Fabric workspace was set up with carefully controlled workspace-level access. Only approved users have workspace access. The security review passes. Six months later, a compliance audit discovers that users with workspace access can query all tables including tables containing PII and financial data that they should not have access to based on their role.

WRONG THINKING

Engineers assumed that workspace-level access control was sufficient because only approved users had workspace access. The distinction between workspace access and table-level data access was not fully understood during security design. Workspace access was treated as the primary and sufficient security boundary.

WHAT IS ACTUALLY HAPPENING

Workspace access in Fabric controls who can see and interact with workspace items. It does not restrict which data within a Lakehouse or Warehouse a user can query once they have workspace access. A user with Viewer access to a workspace can query any table in any Lakehouse in that workspace through the SQL analytics endpoint. For data platforms containing sensitive data, workspace-level access is a necessary but insufficient security control. Data-level security requires additional controls at the table, row, or column level.

THE FIX

Implement data-level security in addition to workspace-level access control. For Lakehouses, use OneLake data access roles to restrict which users can read specific tables. For Warehouses, implement column-level security and row-level security on sensitive tables. For Power BI, implement semantic model RLS for report-layer restrictions. Document the full security model showing which controls operate at which layer and what each control enforces. Test the security model by attempting to access sensitive data with a non-privileged account and verifying the access is denied at the data layer.

34

SECURITY ARCHITECTURE

You implemented column-level security in the Warehouse. Queries that touch secured columns are 20x slower.

THE SITUATION

Column-level security was implemented in the Fabric Warehouse to restrict access to sensitive columns for certain user roles. The security implementation is correct and access controls work as intended. But queries that reference secured columns from secured users are significantly slower than the same queries from unsecured users, even when the query does not actually return the secured column values.

WRONG THINKING

Engineers assumed that column-level security is a transparent filter that adds negligible overhead since it simply hides column values. The performance impact of column security evaluation on query execution was not anticipated because the security check appeared to be a simple column exclusion.

WHAT IS ACTUALLY HAPPENING

Column-level security in Fabric Warehouse is evaluated at query execution time for every query from a secured user, regardless of whether the secured columns are in the SELECT list. The engine must evaluate the user's security context and apply the column restrictions before building the query plan. For complex column security configurations with many rules or rules that reference dynamic functions like `USER_NAME()`, this evaluation adds measurable overhead to every query. The overhead is proportional to the complexity of the security rules, not to whether the secured columns are actually accessed.

THE FIX

Simplify column-level security rules to use static role-based conditions rather than dynamic user-context functions wherever possible. Test query performance with and without column security enabled for your most frequent query patterns and quantify the overhead before deploying to production. For tables where column security causes unacceptable performance degradation, consider an alternative architecture: create separate views that project only the permitted columns for each user group and grant access to the views rather than the underlying table. This moves the column restriction to the view definition rather than the security evaluation layer.

35

SECURITY ARCHITECTURE

You used personal user credentials for all service connections. When the user left the company every pipeline broke overnight.

THE SITUATION

All Fabric pipeline connections to external sources were created using the personal OAuth credentials of the engineer who built them. The pipelines ran reliably for months. When that engineer left the company and their account was deactivated, every pipeline that used their credentials failed simultaneously overnight. Dozens of reports showed no data the next morning.

WRONG THINKING

Engineers used personal credentials because they were the easiest way to authenticate during development and the pipelines worked. Migrating to service principal credentials was treated as a future task that never got prioritized. The single point of failure that personal credentials represent was not identified as an architectural risk.

WHAT IS ACTUALLY HAPPENING

Personal user credentials in Fabric connections are tied to the individual user's account and OAuth token. When the account is deactivated, the token is revoked and every connection using those credentials immediately fails. In a platform where multiple pipelines share connections, one account deactivation can simultaneously break dozens of pipelines across multiple workspaces. Personal credentials also create audit complexity because pipeline actions appear in logs under a personal identity rather than a service identity, making automated access reviews harder.

THE FIX

Replace all personal user credentials in Fabric connections with service principal credentials before the platform goes into production. Create a dedicated service principal for each logical workload domain with the minimum required permissions for that domain. Service principal credentials do not expire when individuals leave the company. Document every service principal, its permissions, and the team responsible for its management. Rotate service principal secrets on a defined schedule and store them in Azure Key Vault rather than in connection configurations directly.

36

SECURITY ARCHITECTURE

You applied security at the Gold layer only. A data breach happened through direct Bronze table access.

THE SITUATION

A security architecture was designed with the assumption that only Gold tables needed access controls because only Gold data was intended for external consumption. Bronze and Silver Lakehouses were left with broad access for engineering teams. A security incident occurred when an engineer with Bronze access extracted a large volume of raw PII data that had not yet been anonymized in the transformation pipeline.

WRONG THINKING

Engineers assumed that security on the serving layer was sufficient because that was the only layer consumers were supposed to use. The risk from internal users with engineering access to raw layers was not included in the threat model. Internal access was treated as implicitly trusted because the users were employees.

WHAT IS ACTUALLY HAPPENING

A security architecture that only protects the Gold serving layer assumes that all threats come from external consumers. Internal threats, whether accidental or intentional, originate from users with engineering access to the raw layers. Raw Bronze data often contains the most sensitive version of data before masking, anonymization, or aggregation has been applied. Leaving Bronze accessible to all engineers because they need it for their work creates a wide attack surface for data exfiltration. The absence of access logs and alerts on Bronze access makes detection slow.

THE FIX

Apply security controls at every layer proportional to the sensitivity of the data at that layer. Bronze tables containing PII or financial data require the same access restrictions as Gold tables regardless of their intended use. Implement OneLake access roles on Bronze Lakehouses to restrict access to only the pipelines and engineers who require it for legitimate ingestion and transformation work. Enable access logging on sensitive tables at all layers and configure alerts for unusually large data reads that may indicate exfiltration. Never treat any layer as implicitly trusted simply because it is internal.

37

SECURITY ARCHITECTURE

You designed a single service principal for all Fabric workloads. One compromised credential gives access to everything.

THE SITUATION

To simplify credential management, a single service principal was created and granted broad permissions across all Fabric workspaces and data sources. All pipelines, notebooks, and automated processes use this one identity. Credential management is simple because there is only one secret to rotate. A security audit flags the architecture as a single point of compromise risk.

WRONG THINKING

Engineers chose a single service principal for operational simplicity. Managing one credential is easier than managing many. The blast radius of a compromised credential was not considered as an architectural risk because the probability of compromise felt low. Simplicity was prioritized over least-privilege design.

WHAT IS ACTUALLY HAPPENING

A single service principal with broad permissions across all workspaces means that compromising one credential gives an attacker access to every data asset on the platform. The blast radius of a credential compromise is directly proportional to the permissions of the compromised identity. In a platform where all workloads share one identity, a phishing attack, a leaked secret in a code repository, or a misconfigured pipeline log that exposes credentials gives full platform access. Broad permissions also make audit logging harder because all automated activity appears under one identity regardless of which workload generated it.

THE FIX

Create separate service principals per workload domain with permissions scoped to only the resources that domain requires. A Bronze ingestion service principal needs read access to external sources and write access to Bronze Lakehouses only. A Gold transformation service principal needs read access to Silver and write access to Gold only. A reporting service principal needs read access to Gold and semantic models only. This limits the blast radius of any individual credential compromise to one domain. Store all secrets in Azure Key Vault and never hardcode them in pipeline configurations or notebook code.

06

FABRIC ENGINEERING PATTERNS

Chapter 6 CI/CD and Environments

Seven scenarios covering how the absence of deployment automation, version control, and environment strategy creates fragile platforms where change becomes increasingly risky over time.

38

CI/CD AND ENVIRONMENTS

You have no deployment pipeline. Every change goes directly to production and rollback requires manual reconstruction.

THE SITUATION

The Fabric platform was built and is maintained by making changes directly in the production workspace. There is no development environment and no deployment pipeline. Changes are tested mentally or in a scratch notebook before being applied. When a breaking change is deployed, rolling back requires manually undoing the change from memory because there is no version history and no previous state to restore.

WRONG THINKING

Engineers assumed that for a small team, the overhead of setting up dev and prod environments and a deployment pipeline was not justified. Direct edits to production felt faster and more efficient. The risk of an unrecoverable breaking change was considered low because the team is careful.

WHAT IS ACTUALLY HAPPENING

Without a deployment pipeline, every change to a production artifact is an uncontrolled release. There is no review step, no testing environment, and no rollback mechanism beyond the engineer's ability to manually reconstruct the previous state. As the platform grows in complexity, the cognitive overhead of tracking what changed and how to revert it grows with it. A platform with no deployment pipeline is one incident away from a situation where the recovery path is unclear and the time to restore is measured in hours rather than minutes.

THE FIX

Establish a minimum viable deployment process: a dev workspace for development and testing, Git integration for version control of Fabric artifacts, and a Fabric deployment pipeline for promoting tested changes to production. Even a simple two-stage dev-to-prod pipeline with manual approval before promotion provides the review step, the version history, and the rollback capability that direct production edits eliminate. Start with the most critical pipelines and semantic models and extend the deployment process to all artifacts over time.

39

CI/CD AND ENVIRONMENTS

You built a full dev, test, and prod environment structure. Keeping three environments in sync is taking more time than building features.

THE SITUATION

Following best practices, a full three-environment structure was implemented: development, test, and production. Each environment has its own workspaces, Lakehouses, and capacity. Deployment pipelines promote changes through each stage. After six months, the team spends more time managing environment drift, synchronizing configurations, and resolving promotion failures than building new data capabilities.

WRONG THINKING

Engineers followed enterprise CI/CD best practices and built a three-environment structure from the start. The overhead of maintaining three environments was not anticipated for a team of the current size. The assumption was that more environments equals more rigor.

WHAT IS ACTUALLY HAPPENING

Three environments is the right architecture for large teams with formal QA processes and regulatory requirements. For small data engineering teams, three environments creates more overhead than value. Each environment must be kept in sync with the others, each has its own configuration drift risk, and each promotion step requires validation and sign-off. When the team is too small to staff each environment stage with dedicated validation resources, the test environment becomes a bottleneck that slows delivery without providing proportional quality assurance.

THE FIX

Match the number of environments to the actual team size and risk profile. A two-environment structure with dev and prod is sufficient for most small and medium data engineering teams. Add a test or staging environment only when the team has the capacity to use it meaningfully for validation rather than just as a pass-through step. Reduce environment maintenance overhead by automating configuration synchronization through infrastructure-as-code and deployment pipeline parameters rather than managing each environment manually.

40

CI/CD AND ENVIRONMENTS

You use Fabric deployment pipelines for promotion. Connection strings are not remapped and prod points at dev data after every promotion.

THE SITUATION

Fabric deployment pipelines were implemented to promote artifacts from dev to prod. Each promotion runs successfully according to the pipeline status. But after every promotion, the semantic model and several pipeline activities in production are found to be pointing at dev data sources rather than production ones. Engineers manually fix the connections after each promotion, and the fix is overwritten by the next promotion.

WRONG THINKING

Engineers assumed that deployment pipelines automatically handle environment-specific configuration. After the first promotion failure, the manual connection fix was applied and the problem seemed solved until the next promotion. The root cause in the deployment pipeline parameter configuration was not investigated because the manual fix worked.

WHAT IS ACTUALLY HAPPENING

Fabric deployment pipelines promote artifact definitions from source to target workspace. They do not automatically remap data source connections unless deployment pipeline parameters are explicitly configured to perform the remapping. If parameters are not configured, the promoted artifact retains the source workspace connection references. Every subsequent promotion overwrites the manually applied fix with the source workspace definition again. The manual fix is not a solution. It is a symptom that the deployment pipeline parameters are not configured.

THE FIX

Configure deployment pipeline parameters that map each dev data source to its prod equivalent. In the Fabric deployment pipeline settings, define parameter rules that replace dev Lakehouse names, Warehouse connection strings, and semantic model data source references with their production counterparts during promotion. Test the parameter mapping by running a promotion and verifying all connections before any production workloads use the promoted artifacts. Document the parameter mapping as part of the deployment pipeline configuration and review it whenever a new data source is added to any promoted artifact.

41

CI/CD AND ENVIRONMENTS

You store Fabric notebook code only in the workspace. When a notebook is overwritten there is no version history.

THE SITUATION

A data engineer accidentally overwrote a production notebook by saving changes from a development session to the wrong notebook. The previous version of the notebook contained complex transformation logic that took weeks to develop. There is no version history in the workspace because Git integration was never configured. The notebook must be reconstructed from memory and from the engineer's local browser cache.

WRONG THINKING

Engineers assumed that Fabric workspace storage was sufficient for notebook persistence because it was always available and accessible. Git integration was considered an optional enhancement for teams that wanted it, not a baseline requirement for protecting production code assets.

WHAT IS ACTUALLY HAPPENING

Fabric workspaces do not maintain automatic version history for notebooks by default. Each save overwrites the previous version and there is no built-in undo beyond the current session. Without Git integration, a notebook overwrite is permanent. The workspace stores the current state of each artifact but provides no mechanism to recover a previous state after an overwrite. This is equivalent to developing software with no source control: individually manageable for a single developer but increasingly risky as the team and codebase grow.

THE FIX

Enable Git integration for all Fabric workspaces that contain production artifacts. Connect the workspace to an Azure DevOps or GitHub repository and configure automatic or manual commit workflows that capture notebook state at defined checkpoints. With Git integration enabled, every saved version of a notebook is recoverable from the repository history. Establish a branching strategy: development work happens on feature branches, and merges to the main branch trigger deployment pipeline promotions to production. Treat notebook code with the same version control discipline as application code.

42

CI/CD AND ENVIRONMENTS

Your CI/CD pipeline deploys all Fabric artifacts together. A change to one notebook triggers a full redeployment of everything.

THE SITUATION

A deployment pipeline was set up to deploy all Fabric artifacts together as a single deployment unit. Every change, regardless of scope, triggers a full deployment of all pipelines, notebooks, semantic models, and Lakehouses in the workspace. A minor fix to one notebook causes a 45-minute full deployment that disrupts all active users and requires a maintenance window.

WRONG THINKING

Engineers built a single deployment unit for simplicity and to ensure that all artifacts are always in a consistent state with each other. The deployment time and disruption cost of deploying all artifacts for every change was not anticipated when the deployment pipeline was designed.

WHAT IS ACTUALLY HAPPENING

A monolithic deployment that includes all artifacts regardless of what changed optimizes for deployment consistency at the expense of deployment frequency and speed. When deployment is slow and disruptive, engineers batch changes to reduce deployment frequency, which increases the risk of each deployment and reduces the team's ability to respond quickly to incidents. The consistency guarantee of a monolithic deployment is valuable but it does not require all artifacts to be deployed simultaneously. Artifacts that have not changed do not need to be redeployed.

THE FIX

Decompose the deployment pipeline into independent deployment units grouped by logical domain or change frequency. Notebooks and pipelines that change frequently should be deployable independently of semantic models and Lakehouses that change rarely. Use Git-based change detection to identify which artifacts changed in each commit and deploy only the affected units. Reserve full environment deployments for major releases or environment provisioning. The goal is to make small changes deployable in minutes rather than requiring a full maintenance window for every fix.

43

CI/CD AND ENVIRONMENTS

You have separate workspaces for dev and prod but a shared Lakehouse. A dev pipeline is writing to the data at scale Lakehouse.

THE SITUATION

Dev and prod workspaces were correctly separated. However, to avoid duplicating large datasets, a single shared Lakehouse was created that both workspaces access. A developer testing a pipeline in the dev workspace ran a write operation that targeted the shared Lakehouse, overwriting data that the production workspace was actively using. The shared Lakehouse was intended for read access only but no technical enforcement prevented writes.

WRONG THINKING

Engineers assumed that developers would know to treat the shared Lakehouse as read-only. The intent was communicated informally during onboarding. No technical control prevented write access from the dev workspace to the shared Lakehouse because the same service principal was used for both environments.

WHAT IS ACTUALLY HAPPENING

A shared resource accessed by both dev and prod environments has no environment boundary. Any workload in either workspace can interact with the shared resource as long as the credentials have access. Informal read-only conventions are not enforced by the platform and will eventually be violated, whether by accident, misunderstanding, or a new team member who was not present for the original convention discussion. A shared Lakehouse between dev and prod is not a dev/prod separation. It is a single environment with two labels.

THE FIX

Remove write access from the dev workspace service principal to any Lakehouse that prod workloads depend on. If a shared dataset is needed for dev testing, create a dev copy of the relevant tables using shortcuts to the prod Lakehouse for read access, and a separate dev Lakehouse for any data that dev pipelines write. The dev workspace should never have write access to any storage that prod pipelines read from. Enforce this at the service principal permission level, not through informal convention.

44

CI/CD AND ENVIRONMENTS

Your deployment pipeline promoted changes correctly. The semantic model refresh did not run and users see stale data.

THE SITUATION

A deployment pipeline successfully promoted a semantic model update from dev to prod. The promotion status showed green. Engineers confirmed the updated semantic model definition was in the production workspace. Business users accessing reports after the deployment still saw data from before the promotion. The semantic model had not been refreshed after promotion and was still framed to the previous Delta table version.

WRONG THINKING

Engineers treated a successful deployment pipeline promotion as equivalent to a successful release. The distinction between deploying an artifact definition and the artifact being in a ready-to-serve state was not understood. A semantic model that has been promoted but not refreshed is not serving current data.

WHAT IS ACTUALLY HAPPENING

Fabric deployment pipelines promote the semantic model definition, schema, and configuration. They do not automatically trigger a semantic model refresh after promotion. A Direct Lake semantic model must be refreshed after promotion to reframe against the current Delta table state in the production Lakehouse. Until the refresh completes, the promoted model serves data from the framing point established before the promotion. Promotion and refresh are two separate operations that must both complete before the release is live.

THE FIX

Add a mandatory semantic model refresh step at the end of every deployment pipeline run that promotes a semantic model. Automate the refresh using the Power BI REST API triggered from a post-deployment script or a Fabric notebook activity. Define the deployment as complete only after the refresh confirms success, not after the promotion step alone. Add this to the deployment checklist as a non-optional step and build it into the automated deployment pipeline rather than relying on engineers to remember it manually after each promotion.

07

FABRIC ENGINEERING PATTERNS

Chapter 7 Cost Architecture

Six scenarios covering where Fabric costs accumulate silently and how architecture decisions that look like optimizations frequently produce the opposite result on the capacity bill.

45

COST ARCHITECTURE

Your Fabric capacity bill doubled after onboarding a new team. You have no visibility into which workloads are consuming capacity.

THE SITUATION

A new data engineering team was onboarded to the Fabric platform and assigned to an existing workspace on the same capacity. The following month the capacity bill was significantly higher than budgeted. Engineers know the new team's workloads are likely responsible but cannot determine which specific pipelines, notebooks, or queries are consuming the most capacity units because there is no per-workload cost tracking.

WRONG THINKING

Engineers assumed that Fabric's shared capacity model would naturally balance workloads and that the platform could absorb additional teams without formal capacity planning. Cost monitoring was treated as a finance concern rather than an engineering concern. No baseline capacity consumption was established before the new team was onboarded.

WHAT IS ACTUALLY HAPPENING

Fabric capacity is billed based on total capacity unit consumption across all workspaces assigned to that capacity. Without per-workspace or per-item consumption tracking, it is impossible to attribute cost increases to specific workloads after the fact. The Fabric Monitoring Hub provides capacity unit consumption data but requires proactive configuration and baselining to be useful for cost attribution. Onboarding a new team without measuring their workload's capacity impact before and after onboarding makes cost accountability impossible.

THE FIX

Before onboarding any new team or workload to a shared capacity, establish a consumption baseline using the Fabric Monitoring Hub. Record the average daily capacity unit consumption by workspace for the two weeks before onboarding. After onboarding, compare consumption by workspace to identify the incremental cost of the new workloads. Assign each workspace to a logical cost center and review consumption per workspace monthly. If a single workspace consistently consumes a disproportionate share of capacity, investigate and optimize the specific workloads responsible before they become a budgetary problem.

46

COST ARCHITECTURE

You chose the lowest Fabric capacity tier. Pipelines are queuing and reports are timing out during business hours.

THE SITUATION

To minimize cost, the smallest available Fabric capacity tier was selected. The platform worked correctly during development and testing with a small team. After go-live with full business user load, morning ETL pipelines are queuing behind each other, Power BI reports are timing out during peak hours, and interactive notebook sessions are slow. Engineers are constantly context-switching to manage pipeline queues instead of building new capabilities.

WRONG THINKING

Engineers assumed that the smallest tier was sufficient because the platform worked during development. Development load with a handful of engineers testing occasionally is not representative of production load with dozens of concurrent users and multiple simultaneous pipeline runs. The capacity decision was made based on development observation rather than production load modeling.

WHAT IS ACTUALLY HAPPENING

Fabric capacity tiers determine the maximum number of concurrent capacity units available to all workloads. Under-sized capacity creates a queue for every workload that exceeds the available units. Pipelines wait for units to become available, interactive queries are deprioritized behind batch workloads, and the overall platform throughput is capped below what the business requires. The cost of under-sizing is paid in engineer time managing queues and in business user productivity lost to timeouts, not just in the monthly capacity bill.

THE FIX

Model production load before selecting a capacity tier. Estimate concurrent pipeline count, average Spark job size, peak report user count, and semantic model refresh frequency. Use the Fabric capacity sizing guidance to translate these estimates into required capacity units. Select a tier that provides headroom above the estimated peak load, typically 20 to 30 percent buffer. If production load is difficult to estimate before go-live, start one tier above the minimum and monitor actual consumption in the first month. It is easier to downsize after measuring actual consumption than to recover from chronic under-capacity.

47

COST ARCHITECTURE

Your Fabric capacity is bursting constantly. You are being charged for burst capacity that was not budgeted.

THE SITUATION

A Fabric capacity was correctly sized for average load. But the monthly bill consistently includes burst charges that were not in the budget. The engineering team was not aware that Fabric capacity can burst beyond the purchased tier and that burst consumption is billed separately. The burst charges appear as unexpected line items each month.

WRONG THINKING

Engineers assumed that a Fabric capacity tier provided a hard ceiling on consumption and cost. The concept of capacity bursting and its billing implications was not understood when the capacity was provisioned. The monthly bill appeared straightforward until the burst charges appeared.

WHAT IS ACTUALLY HAPPENING

Fabric capacity supports bursting which allows workloads to temporarily consume capacity units beyond the purchased tier limit. Burst consumption is billed at a higher rate than base capacity. Bursting occurs when multiple workloads compete for capacity units simultaneously and demand exceeds the base tier allocation. A capacity that is correctly sized for average load but not for peak concurrent load will burst during peak periods. If peaks are frequent, burst charges can add significantly to the monthly bill beyond the base capacity cost.

THE FIX

Review the Fabric Monitoring Hub to identify which time periods and which workloads are triggering burst consumption. If bursting is caused by predictable morning ETL peaks coinciding with business user report access, stagger the workload schedules to spread peak demand across a wider time window. If bursting is caused by a consistently under-sized capacity, evaluate whether upgrading the base tier costs less than the recurring burst charges. Enable capacity burst alerts to notify the engineering team when burst thresholds are crossed so unexpected charges can be investigated immediately rather than discovered on the monthly bill.

48

COST ARCHITECTURE

You store all data forever in OneLake. Storage costs are growing linearly with no data lifecycle policy.

THE SITUATION

The Fabric platform has been running for two years with no data retention or archival policy. All data from all sources across all Medallion layers is retained indefinitely. OneLake storage costs are growing at the same rate as data ingestion with no offset from deletion or archival. The engineering team is aware the storage bill is growing but has not prioritized implementing a lifecycle policy.

WRONG THINKING

Engineers assumed that storage is cheap and that retaining all data is safer than deleting it because historical data might be needed for audits, reprocessing, or debugging. The cumulative cost of two years of unbounded retention was not calculated. Each individual day's data addition seemed trivially small.

WHAT IS ACTUALLY HAPPENING

OneLake storage is billed per gigabyte per month. Two years of retained data with no deletions means the storage bill is the sum of every day's data ingestion across the entire platform lifetime. For platforms ingesting significant daily volumes, this accumulates to substantial monthly costs that grow permanently without intervention. Beyond cost, indefinite retention creates compliance risks when data has legally required deletion obligations and operational risks when debugging tools must search through years of unnecessary history.

THE FIX

Define a data retention policy for each Medallion layer based on business and regulatory requirements. Bronze data typically has a short retention window of 30 to 90 days since it is replaced by validated Silver data. Silver data retains for a medium window aligned with reprocessing requirements. Gold data retains for the business reporting horizon. Implement the policy using Delta table time travel expiration by running `VACUUM tablename RETAIN X HOURS` on a schedule. For data with legal retention requirements, archive to lower-cost storage rather than retaining in hot OneLake storage. Calculate and track monthly storage cost per layer to measure the impact of the retention policy.

49

COST ARCHITECTURE

You run OPTIMIZE and VACUUM on all tables daily regardless of whether they changed. Maintenance is consuming capacity with no benefit.

THE SITUATION

A maintenance pipeline runs OPTIMIZE and VACUUM on every table in every Lakehouse every night. The pipeline was set up when the platform launched to keep Delta tables healthy. After two years, the platform has hundreds of tables but most of them are reference tables or slowly changing dimensions that receive updates once a week or less. The nightly maintenance pipeline consumes significant capacity running optimization on tables that have not changed since the last maintenance run.

WRONG THINKING

Engineers assumed that running OPTIMIZE and VACUUM on all tables daily was a safe default that guaranteed table health. The cost of running maintenance on tables that do not need it was not considered because each individual table's optimization appeared to be a small operation. The aggregate cost of unnecessary maintenance across hundreds of tables was not measured.

WHAT IS ACTUALLY HAPPENING

OPTIMIZE compacts small Delta files and applies V-Order. VACUUM removes files no longer referenced by the transaction log. Both operations consume capacity units proportional to the table size and the number of files processed. Running OPTIMIZE on a table that received no new writes since the last optimization is a no-op in terms of data improvement but still consumes capacity to scan the table and confirm there is nothing to compact. For a platform with hundreds of tables, daily maintenance on all tables is a significant recurring capacity cost that provides no benefit for tables that have not changed.

THE FIX

Implement change-aware maintenance that runs OPTIMIZE and VACUUM only on tables that received writes since the last maintenance run. Track the last write timestamp for each table in a maintenance control table and skip maintenance for tables where the last write predates the last maintenance run. For large fact tables that receive daily loads, continue running OPTIMIZE after every load as part of the load pipeline rather than as a separate nightly job. For slowly changing reference tables, run maintenance weekly rather than daily. Measure capacity unit consumption of the maintenance pipeline before and after this change to quantify the savings.

50

COST ARCHITECTURE

Your most expensive pipeline runs every 15 minutes. The business actually only needs the data refreshed once a day.

THE SITUATION

A data ingestion pipeline was configured to run every 15 minutes to provide near-real-time data freshness. The pipeline consumes significant capacity with 96 runs per day. During a business review, it is discovered that the reports powered by this pipeline are only opened once in the morning and once before end of day. The business users have never needed 15-minute freshness and were not aware the pipeline was running at that frequency.

WRONG THINKING

Engineers assumed that more frequent refreshes are always better and that the business would appreciate near-real-time data even if they did not explicitly request it. The capacity cost of 96 daily pipeline runs versus 1 was not calculated. Freshness frequency was treated as a technical decision rather than a business requirement that should be explicitly agreed upon.

WHAT IS ACTUALLY HAPPENING

Pipeline scheduling frequency is a direct cost multiplier. A pipeline that runs every 15 minutes consumes 96 times the capacity of the same pipeline running once a day, assuming similar execution characteristics. For pipelines that move or transform data, each run also incurs storage write costs and downstream refresh costs for any semantic models that refresh after each pipeline completion. Running a pipeline at higher frequency than the business requires does not deliver more value. It delivers the same value at a higher cost.

THE FIX

Review the scheduling frequency of every pipeline against the actual business requirement for data freshness. For each pipeline, identify the consuming reports and dashboards and ask the business users how often they genuinely need the data to refresh. Align the pipeline schedule to the business requirement rather than to a technical default. For pipelines where freshness requirements are genuinely uncertain, start with a daily schedule and increase frequency only when a specific business case justifies the additional cost. Document the business justification for every pipeline that runs more frequently than daily so the decision can be reviewed when capacity costs are audited.

CONCLUSION

Architecture debt compounds silently.

Across 50 scenarios, one pattern repeats. Architecture mistakes do not announce themselves at the moment they are made. They announce themselves months later when the platform is carrying production workloads, teams have grown, and the cost of changing the foundation has multiplied. The most reliable way to avoid the patterns in this book is to ask one question before each architecture decision: what does this look like when the platform is ten times larger than it is today?

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.

Lakehouse and PySpark

> Your Delta table has millions of small files. OPTIMIZE is making it worse.
>
> Your notebook runs in 2 minutes alone. 20 minutes when triggered from a pipeline.

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.

CONNECT

🌐 www.ssanjaychandra.com

💼 linkedin.com/in/ssanjaychandra

✉️ ssanjaychandra@outlook.com

Sanjay Chandra
The Databricks + Fabric guy on LinkedIn