

When the Lakehouse Works, but the Operating Model Breaks... Especially On-Premises

Julien Laurenceau | Published 2026-05-04 | Updated 2026-10-01

<https://pepitedata.com/when-the-lakehouse-works-but-the-operating-model-breaks-especially-on-premises/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



Note: This article was inspired by a [LinkedIn post](#) by **Can Sinan A.** on the hidden operational cost of an Iceberg migration, especially when governance moves from one access-control plane to several layers: catalog, storage and engine configuration

Introduction

Many organizations are modernizing their data platforms with lakehouse architectures, and most of them evaluate the architecture while too few evaluate the operating model. Apache Iceberg and Delta Lake make open formats, separated storage and compute, and multi-engine access look like an obvious win. The hidden trap is that the lakehouse may work technically while the operating model is not ready, and on-premises organizations feel that gap most.

The promise is attractive: keep data in open formats, separate storage and compute, avoid unnecessary vendor lock-in, support analytics and AI workloads, and let multiple engines query the same datasets.

Apache Iceberg and Delta Lake are two of the most common table formats behind this movement. Iceberg is an open table format for large analytical datasets and is designed to work with engines such as Spark, Trino, Presto, Flink, Hive and Impala. Delta Lake provides ACID transactions, scalable metadata handling, and unified batch and streaming processing on top of existing data lakes such as S3, ADLS, GCS and HDFS.

On paper, the migration can look like an obvious win.

Lower storage cost.

More flexibility.

More interoperability.

More control over data.

Less dependency on a single warehouse.

But there is a hidden trap.

Most companies evaluate the **architecture**.

Too few evaluate the **operating model**.

And for on-premises organizations, this problem is even more important.

A managed cloud platform can absorb part of the operational complexity. An on-premises platform usually cannot. Your internal team owns much more of the stack: storage, catalog, permissions, engines, metadata maintenance, monitoring, backup, recovery and incident response.

The lakehouse may work technically.

But the operating model may not be ready.

You changed more than the table format

You changed more than the table format: you split access control, metadata management, query execution, table optimization, auditability, backup and recovery across table format, catalog, storage, engines, identity, orchestration and monitoring. A traditional data warehouse hides those responsibilities inside one platform. Each lakehouse layer may be excellent individually, but your team now owns the seams between them, and that is where governance and reliability incidents happen.

Access control, metadata management, query execution, table optimization, auditability, storage permissions, backup, recovery and incident response are often handled by the same product.

A composable lakehouse is different.

The responsibilities are split across multiple layers:

- table format;
- catalog;
- object storage or distributed storage;
- query engines;
- IAM or identity system;
- orchestration tools;
- governance layer;
- monitoring;

- maintenance jobs;
- backup and recovery procedures.

Each layer may be excellent individually.

The problem is that your team now owns the seams between them.

That is where governance and reliability incidents happen.

A user may lose access in the catalog but still retain direct access to storage.

One engine may enforce permissions differently from another.

A service account may keep broader privileges than intended.

A table may work in Spark but fail in Trino because metadata, versions or configuration drifted.

A catalog may point to one table state while storage permissions allow another path.

The architecture works.

But the operating model is fragile.

Why metadata matters in open table formats

Metadata is what makes an open table format work: it is the layer that tells engines which files belong to the table, which schema is valid and which version should be read, and the catalog is the system that helps engines find it. That layer enables transactions, time travel, schema evolution, concurrent reads and writes, and more efficient query planning. If metadata becomes the source of truth, then metadata operations become production operations.

In a lakehouse, a table is not simply a folder full of Parquet files.

A table is made of two things:

Data files	= the physical files stored in object storage or distributed storage
Metadata	= the layer that tells engines which files belong to the table
Table version	= the valid state of the table at a given point in time
Catalog	= the system that helps engines find the current table metadata

This metadata layer is what makes modern table formats powerful.

It enables transactions, time travel, schema evolution, concurrent reads and writes, and more efficient query planning.

But it also explains why operational readiness matters.

If metadata becomes the source of truth, then metadata operations become production operations.

How Delta Lake stores table metadata

Delta Lake uses a transaction log stored inside the table directory, usually in a folder called `_delta_log`. The log records committed transactions, so it tells the engine which files were added or removed, which schema is valid and which table version should be read, with periodic checkpoints alongside the JSON commits. Teams running it must therefore care about log retention, checkpoints, vacuum policies, concurrent writes, storage permissions and engine compatibility.

This log records committed transactions on the table. In practical terms, it tells the engine which files were added, which files were removed, which schema is valid, and which table version should be read.

A simplified Delta table looks like this:

```
/my_table/
  part-0001.parquet
```

```
part-0002.parquet
part-0003.parquet

_delta_log/
  00000000000000000000000000000000.json
  00000000000000000000000000000001.json
  00000000000000000000000000000002.json
  ...
  checkpoint.parquet
```

Delta Lake documentation describes the transaction log as the table metadata stored in the `_delta_log` directory, and Databricks describes the transaction log as central to ACID transactions, scalable metadata handling and time travel.

The operational consequence is simple: teams must care about log retention, checkpoints, vacuum policies, concurrent writes, storage permissions and engine compatibility.

Delta can be robust.

But self-managed Delta is still something you operate.

It is not the same thing as Delta managed inside Databricks.

How Apache Iceberg stores table metadata

Apache Iceberg uses a more hierarchical metadata model: instead of one linear transaction log, it tracks table state through metadata files, snapshots, manifest lists and manifest files, with the catalog pointing engines to the current metadata file. Snapshots represent versions of the table and manifests track which data files belong to them. The design gives strong multi-engine flexibility, and that freedom must be operated.

Instead of relying primarily on a linear transaction log, Iceberg tracks table state through metadata files, snapshots, manifest lists and manifest files.

A simplified Iceberg table looks like this:

```
/my_table/
  data/
    part-0001.parquet
    part-0002.parquet
    part-0003.parquet

  metadata/
    v1.metadata.json
    v2.metadata.json
    snap-001.avro
    manifest-list-001.avro
    manifest-001.avro
```

The catalog points engines to the current metadata file.

The metadata file describes the table state.

Snapshots represent versions of the table.

Manifests track which data files belong to those snapshots.

Snowflake's Iceberg documentation explains that an Iceberg catalog stores the current metadata pointer for tables, mapping a table name to the current metadata file location, and must support atomic operations to update that pointer. Trino's Iceberg documentation also describes the table metadata file as tracking schema, partitioning, table properties and snapshots.

This gives Iceberg strong flexibility, especially for multi-engine environments.

But it also creates operational responsibilities:

- catalog ownership;
- snapshot expiration;
- manifest rewrite;
- orphan file cleanup;
- metadata retention;
- compatibility between engines;
- storage permission alignment;
- monitoring of table maintenance jobs.

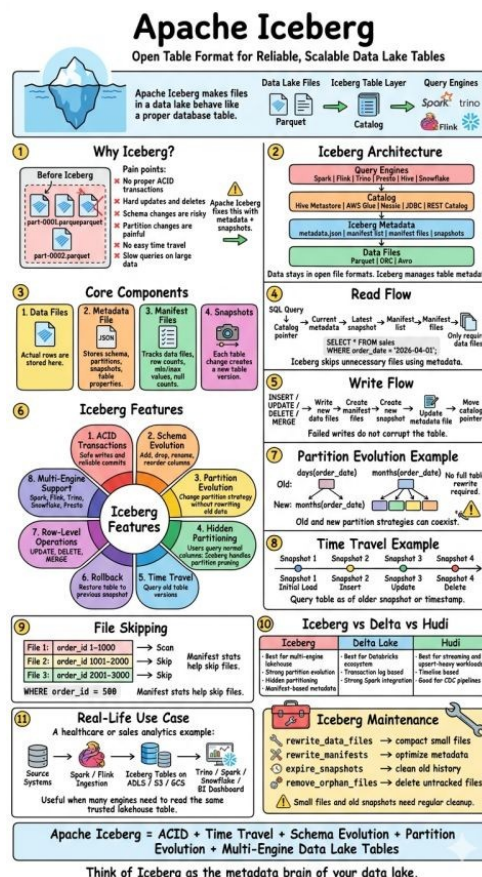
Iceberg is not worse than Delta.

It is more explicit.

It gives you more architectural freedom, but that freedom must be operated.

Cheat Sheet by Arvind D

The cheat sheet below is Arvind D's one-page map of Apache Iceberg, published on LinkedIn and linked in the heading. It covers the core components (data files, metadata files, manifest files, snapshots), the architecture, the read and write flows, time travel, partition evolution, file skipping and table maintenance. A compact Iceberg versus Delta Lake versus Hudi comparison sits at the bottom, next to the maintenance commands that keep tables healthy.



Special case: what if your company is mostly on-premises?

If your company is mostly on-premises, the Iceberg versus Delta question is secondary: the real question is whether you are ready to operate the lakehouse control plane yourselves. For cloud-native companies the discussion often becomes a comparison between managed platforms; for you it is a distributed platform engineering project, not a simple storage migration. Iceberg and Delta are not bad technologies, but they expose database-like responsibilities the organization may not be staffed to operate.

For on-premises companies, the question is different.

It is not mainly:

Should we use Snowflake or Databricks?

It is:

Are we ready to operate the lakehouse control plane ourselves?

This changes the risk profile completely.

In an on-premises architecture, the stack may look like this:

Storage	= MinIO, Ceph, HDFS, NetApp StorageGRID, Dell ECS, etc.
Table format	= Apache Iceberg or Delta Lake
Catalog	= Hive Metastore, Iceberg REST Catalog, Nessie, Polaris, JDBC catalog
Engines	= Spark, Trino, Flink, Presto, Dremio, StarRocks, etc.
Security	= LDAP, Active Directory, Kerberos, Ranger, SSO, storage ACLs
Orchestration	= Airflow, Dagster, Argo, Control-M, custom jobs
Monitoring	= Prometheus, Grafana, logs, vendor tools

This can work.

But it is not a simple storage migration.

It is a distributed platform engineering project.

In a managed cloud platform, several responsibilities may be absorbed by the vendor: catalog management, table optimization, governance integration, monitoring, storage integration and support.

In an on-premises lakehouse, the internal team usually owns much more of the stack.

That includes:

- storage configuration;
- network performance;
- catalog availability;
- authentication and authorization;
- query engine configuration;
- metadata maintenance;
- backup and recovery;
- audit trails;
- upgrade compatibility;
- incident response.

This is why on-premises lakehouse projects should be evaluated differently.

The risk is not that Iceberg or Delta are bad technologies.

The risk is that they expose database-like responsibilities that the organization may not be staffed to operate.

The real comparison: format versus operating model

The real comparison is not Iceberg versus Delta but self-managed operation versus platform-managed operation. Four cases follow: Iceberg self-managed, Iceberg managed, Delta self-managed, Delta managed, with operational burden highest on self-managed Iceberg and lowest on managed Delta. The discussion is often framed as:

Should we choose Iceberg or Delta?

That is the wrong first question.

A better question is:

Are we operating the lakehouse ourselves, or is a platform absorbing part of the complexity?

The real comparison has four cases:

1. Iceberg self-managed
2. Iceberg managed, for example Snowflake
3. Delta self-managed
4. Delta managed, for example Databricks

Dimension	Iceberg self-managed	Iceberg managed, e.g. Snowflake	Delta self-managed	Delta managed, e.g. Databricks
Typical setup	Iceberg tables on storage, with separately managed catalog, engines, permissions and maintenance jobs	Iceberg tables governed and operated through a managed platform such as Snowflake	Delta tables on storage, usually with Spark or another engine, plus custom governance and maintenance	Delta tables governed through Databricks and Unity Catalog
Main benefit	Maximum openness and architectural control	Open table format with more managed operations	Delta transaction model without full platform dependency	Strong integrated governance and operational guardrails
Main operational risk	Highest risk of drift between catalog, storage permissions, engines and maintenance jobs	Lower than self-managed Iceberg, but still depends on platform design and external storage configuration	Medium to high, depending on governance, permissions and maintenance maturity	Lowest operational burden among the four, if the company accepts platform dependency
Governance model	Must be designed across catalog, storage, engines and service accounts	More centralized through the managed platform and catalog	Must be designed around Delta logs, storage access, engines and permissions	Centralized through Unity Catalog for managed tables
Multi-engine flexibility	Strongest	Strong, but mediated by the managed catalog/platform	Possible, but must be validated engine by engine	Possible, but strongest experience remains inside Databricks
Catalog ownership	Internal platform team	Platform/vendor handles part of the burden	Internal platform team or custom metastore setup	Databricks Unity Catalog

Dimension	Iceberg self-managed	Iceberg managed, e.g. Snowflake	Delta self-managed	Delta managed, e.g. Databricks
Table maintenance	Internal responsibility: compaction, snapshots, orphan files, metadata cleanup, small files	Platform can handle lifecycle operations such as compaction	Internal responsibility: optimize, vacuum, file sizing, retention policies	More automated through managed tables and predictive optimization
Object storage / distributed storage debugging	Strong internal skills required	Still useful, but less exposed day to day	Required if self-managed	Less exposed in managed mode
Vendor lock-in	Lowest	Medium: open format, but platform-managed operations	Low to medium	Highest
Best fit	Mature platform teams wanting open architecture and multi-engine control	Teams wanting Iceberg openness without operating everything themselves	Teams already strong on Spark/Delta but not fully committed to Databricks	Teams prioritizing speed, governance, simplicity and managed operations
Worst fit	Teams without platform engineering maturity	Teams expecting "managed" to mean "no ownership"	Teams without clear governance and maintenance ownership	Teams requiring strong vendor neutrality

Snowflake documents that Snowflake-managed Iceberg tables can use Snowflake as the Iceberg catalog and that Snowflake can perform data compaction for Snowflake-managed Iceberg tables. Databricks describes Unity Catalog managed tables as the recommended table type for Delta Lake and Apache Iceberg in Databricks, with Unity Catalog managing read, write, storage and optimization responsibilities; Databricks also documents predictive optimization as automatically running maintenance operations for Unity Catalog managed tables.

So the practical hierarchy is not simply:

Delta is safer than Iceberg.

It is closer to this:

1. **Highest operational burden:** Iceberg self-managed
2. **High operational burden:** Delta self-managed
3. **Reduced operational burden:** Iceberg managed on a platform such as Snowflake
4. **Most integrated operational model:** Delta managed on Databricks with Unity Catalog

This does not mean every company should choose Databricks.

It means that the more self-managed and composable the architecture is, the more explicit the operating model must be.

A simple summary:

Iceberg self-managed gives maximum freedom.

Iceberg managed keeps the open format but reduces some operational burden.

Delta self-managed gives the Delta transaction model but still requires serious platform ownership.

Delta managed gives the strongest guardrails, at the cost of stronger platform dependency.

On-premises comparison

The relevant on-premises comparison is Iceberg self-managed against Delta self-managed against a vendor-supported lakehouse such as Cloudera, Dremio, Starburst or HPE Ezmeral, and for many of my customers it matters far more than Snowflake versus Databricks. A fully self-managed lakehouse is not impossible. But it should be treated like critical infrastructure, not like a side project and not like a simple file format migration.

It is this one:

Dimension	Iceberg on-prem self-managed	Delta on-prem self-managed	On-prem vendor-supported lakehouse
Typical setup	Iceberg + MinIO/Ceph/HDFS + Spark/Trino + Hive/Nessie/REST catalog	Delta + HDFS/MinIO/S3-compatible storage + Spark/Trino	Cloudera, Dremio, Starburst, HPE Ezmeral or similar
Main benefit	Open format, strong multi-engine design, high flexibility	Strong transaction log model, good Spark ecosystem, simpler mental model	More integrated support, governance and operations
Main risk	Highest operational complexity: catalog, metadata, engines, storage permissions	Operational complexity around transaction log, vacuum, concurrent writes and engine compatibility	Vendor dependency, cost, feature limits
Governance	Must be assembled across catalog, storage, engines and identity systems	Must be assembled across storage, engines, logs and identity systems	More centralized, depending on vendor
Metadata operations	Snapshot expiration, manifest rewrite, orphan cleanup, catalog consistency	Checkpoints, log retention, vacuum, file compaction	Partly automated or vendor-assisted
Multi-engine access	Strong design goal	Possible, but must be validated engine by engine	Depends on platform
Required skills	Very high platform engineering maturity	High Spark/lakehouse engineering maturity	Medium to high, depending on customization
Best fit	Teams with strong infrastructure, data platform and SRE skills	Teams standardized on Spark with good operations discipline	Enterprises wanting on-prem control with vendor support
Worst fit	Small teams expecting warehouse-like simplicity	Teams without clear storage/log maintenance ownership	Teams needing full openness and no vendor dependency

For many on-premises companies, a fully self-managed lakehouse is not impossible.

But it should be treated like critical infrastructure.

Not like a side project.

Not like a simple file format migration.

Not like “just put Parquet on S3-compatible storage”.

What can go wrong?

Six failure modes come up repeatedly: permissions defined in too many places drift apart, the catalog becomes a critical dependency, metadata maintenance is ignored, multi-engine access turns into multi-inconsistency, storage skills become production skills, and the cost moves from infrastructure to people. None of them break the architecture on day one. They show up as operational debt weeks or

months later.

1. Governance drift

Permissions are defined in too many places.

Catalog permissions, storage ACLs, service accounts, query engine configurations and BI access rules slowly diverge.

The platform still works, but nobody can confidently answer:

Who can access this data, through which path, and why?

This is one of the most dangerous failure modes because it may remain invisible for weeks or months.

On-premises, the risk is often amplified by legacy identity systems, shared technical accounts, historical storage permissions and inconsistent integration between LDAP, Active Directory, Kerberos, Ranger, engines and storage.

2. The catalog becomes a critical dependency

In open table formats, the catalog is not a minor technical component.

It helps engines locate the current metadata for a table.

For Iceberg, the catalog stores the pointer to the current metadata file and must support atomic updates to that pointer.

If the catalog is unavailable, stale or misconfigured, engines may fail, see inconsistent state, or behave differently depending on their configuration.

For on-premises teams, this raises practical questions:

- Who owns the catalog?
- Is it highly available?
- Is it backed up?
- How is it upgraded?
- Who monitors it?
- Who tests recovery?
- What happens if the catalog and storage disagree?

Catalog ownership is an architecture decision.

Not an implementation detail.

3. Metadata maintenance is ignored

Metadata structures grow over time.

Files are added.

Files are removed.

Snapshots accumulate.

Manifests need maintenance.

Small files hurt performance.

Old versions need retention policies.

Deleted files may need cleanup.

If nobody owns these jobs, performance and cost degrade gradually.

The issue rarely appears on day one.

It appears after the platform has been running for several weeks or months.

This is especially important on-premises, where storage capacity, I/O throughput and network bottlenecks are usually more constrained than in elastic cloud environments.

4. Multi-engine becomes multi-inconsistency

Multi-engine access is one of the strongest promises of open lakehouse architectures.

It is also one of the biggest operational risks.

Different engines may have:

- different table format versions;
- different feature support;
- different caching behavior;
- different permission models;
- different write semantics;
- different release cycles;
- different configuration defaults.

The fact that several engines can read the same table does not mean they are all equally safe to write to it.

Multi-engine should be designed deliberately.

It should not happen accidentally.

5. Storage skills become production skills

A warehouse hides storage complexity.

A lakehouse exposes it.

When something breaks, the team may need to understand:

- storage permissions;
- path conventions;
- lifecycle policies;
- file-level debugging;
- service account behavior;
- replication;
- backup;
- encryption;

- network throughput;
- small-file impact;
- disk and object-store behavior.

For on-premises organizations, this is often the biggest reality check.

The data platform team may need storage engineering skills alongside SQL and Spark skills.

6. The cost moves from infrastructure to people

This is the hidden business case error.

The software may be open source.

The storage may already exist.

The licensing cost may look attractive.

But the platform team may now spend more time operating the stack.

If that human cost is not included in the migration plan, the business case is incomplete.

A cheaper architecture can still be more expensive to operate.

Especially on-premises.

The dangerous sentence: “we will manage it ourselves”

The dangerous sentence for on-premises teams is “We will manage it ourselves.” That claim is as dangerous as “The vendor will handle it” because it must be proven rather than assumed. Operating a lakehouse on-premises requires clear ownership of storage, catalog, table format configuration, identity and permissions, engine configuration, metadata maintenance, monitoring, backup, recovery, incident response, upgrades and cost and capacity planning.

The vendor will handle it.

It is also:

We will manage it ourselves.

That may be true.

But it must be proven.

Operating a lakehouse on-premises requires clear ownership of:

- storage;
- catalog;
- table format configuration;
- identity and permissions;
- engine configuration;
- metadata maintenance;
- monitoring;
- backup;

- recovery;
- incident response;
- upgrades;
- cost and capacity planning.

If those responsibilities are not assigned, tested and monitored, the platform will accumulate operational debt.

The stack may still run.

But it will become harder to debug, harder to secure and harder to trust.

A practical readiness checklist for on-premises teams

The readiness checklist for on-premises teams covers seven areas: storage, catalog, identity and governance, metadata maintenance, reliability, engine compatibility and team maturity. Before migrating critical workloads to Iceberg, Delta or any open table format, ask the questions listed under each of them.

Storage

Which storage backend will be used?

Is it HDFS, MinIO, Ceph, NetApp StorageGRID, Dell ECS, another S3-compatible system, or something else?

Has it been tested with the chosen table format and engines?

What are the throughput and latency characteristics?

How are backups handled?

How is encryption handled?

How are lifecycle policies handled?

Who owns storage incidents?

Catalog

Which catalog will be used?

Hive Metastore?

Iceberg REST Catalog?

Nessie?

Polaris?

JDBC catalog?

Vendor catalog?

Is the catalog highly available?

Is it backed up?

Who owns schema changes?

Who owns catalog upgrades?

Who validates catalog and storage consistency?

Identity and governance

What is the source of truth for identity?

LDAP?

Active Directory?

Kerberos?

SSO?

Ranger?

Engine-level roles?

Storage ACLs?

Can users bypass the catalog and read files directly?

Are service accounts scoped narrowly enough?

Can access be audited centrally?

Do Spark, Trino, Flink and BI tools enforce the same permissions?

Metadata maintenance

Who owns compaction?

Who owns snapshot expiration?

Who owns manifest rewrite?

Who owns orphan file cleanup?

Who owns Delta vacuum and log retention?

Who monitors small files?

Who monitors metadata growth?

Who receives alerts when maintenance jobs fail?

Reliability

What happens if the catalog is down?

What happens if the storage backend is slow?

What happens if one engine writes bad data?

Can the team roll back safely?

Have recovery procedures been tested?

What are the RPO and RTO for critical datasets?

What is the degraded mode?

Engine compatibility

Which engine is allowed to write?

Which engines are read-only?

Which table format features are allowed?

Are all engines compatible with those features?

Who approves upgrades?

How are regressions tested?

How is caching invalidated?

How is concurrency tested?

Team maturity

Can the team inspect table metadata?

Can the team debug storage-level permissions?

Can the team diagnose query engine behavior?

Can the team recover from bad writes?

Can the team explain the difference between table access and storage access?

Can the team operate the system outside the happy path?

Recommended migration approach for on-premises companies

For most on-premises companies, a big-bang lakehouse migration is a bad idea. A safer first version is intentionally boring: one storage backend, one table format, one catalog, one primary compute engine, one governance model, automated maintenance, monitored SLAs and tested recovery, on a single non-critical domain. Multi-engine access should be earned, not assumed, and a good on-premises lakehouse should be boring before it becomes flexible.

A safer first version is intentionally boring:

- one storage backend
- + one table format
- + one catalog
- + one primary compute engine
- + one governance model
- + automated maintenance
- + monitored SLAs
- + tested recovery

Start with one domain.

Not the most critical one.

Choose one primary write engine.

Do not allow every engine to write at the beginning.

Define the catalog strategy.

Define the permission model.

Automate maintenance from day one.

Monitor metadata growth, small files, query planning time and failed commits.

Test recovery before production.

Add multi-engine access only where there is a real business case.

Multi-engine access should be earned, not assumed.

A good on-premises lakehouse should be boring before it becomes flexible.

Recommended phased roadmap

The roadmap runs in four phases: a controlled pilot on one non-critical analytical domain, governance validation, production hardening, and only then careful engine expansion. The pilot optimizes for operational clarity rather than flexibility. Multi-engine access comes last, because it should be earned rather than assumed.

Phase 1: Controlled pilot

Pick one non-critical analytical domain.

Use one storage backend, one catalog, one table format and one primary engine.

Success criteria:

- table creation works;
- read/write works;
- permissions are clear;
- maintenance jobs run;
- recovery is tested;
- monitoring exists.

Do not optimize for flexibility yet.

Optimize for operational clarity.

Phase 2: Governance validation

Before scaling, test access paths.

Can a user bypass the catalog?

Can a service account read raw files?

Do Spark and Trino enforce the same access rules?

Can audit logs answer who accessed what?

Can deleted or restricted data still be queried through an old snapshot?

This phase is not optional.

It is where many future incidents are prevented.

Phase 3: Production hardening

Add:

- high availability for the catalog;
- backup and restore procedures;
- metadata maintenance alerts;
- storage capacity alerts;
- engine compatibility tests;
- incident runbooks;
- upgrade procedures;
- rollback procedures.

At this stage, the platform starts to become a real production system.

Phase 4: Expand engines carefully

Only now should multi-engine access be expanded.

For each engine, define:

- read-only or read-write;
- supported table features;
- permission enforcement model;
- caching behavior;
- upgrade process;
- owner;
- rollback plan.

Do not let “open format” become “uncontrolled access”.

The key lesson

The key lesson is that freedom is not free: the one universal mistake is choosing the architecture without pricing the operating model. Moving from a warehouse-centric architecture to a composable lakehouse buys flexibility, openness and potentially lower infrastructure costs, and it hands you responsibilities that used to be hidden inside a single platform. For on-premises companies, the lakehouse should not be sold as a cheaper warehouse but treated as a platform engineering program.

But freedom is not free.

When you move from a warehouse-centric architecture to a composable lakehouse, you gain flexibility, openness and potentially lower infrastructure costs.

At the same time, you absorb responsibilities that used to be hidden inside a single platform.

For cloud-managed companies, the key question is:

Which platform absorbs which operational responsibilities?

For on-premises companies, the harder question is:

Are we ready to operate the lakehouse control plane ourselves?

Iceberg self-managed gives maximum architectural freedom, but it demands strong platform engineering.

Iceberg managed can reduce part of the operational burden while keeping the open table format.

Delta self-managed can be powerful, but still requires clear ownership of logs, storage, governance and maintenance.

Delta managed gives stronger operational guardrails, but increases platform dependency.

There is no universally correct answer.

But there is one universal mistake:

Choosing the architecture without pricing the operating model.

For on-premises companies, the lakehouse should not be sold as a cheaper warehouse.

It should be treated as a platform engineering program.

If the organization is not ready to operate catalogs, metadata, storage permissions, maintenance jobs, monitoring and recovery, the lakehouse will become fragile regardless of the table format.

A modern lakehouse can be a strong strategic move.

But a modern architecture on an unprepared operating model is technical debt with better marketing.

Update 2026-09-29. This post is now part of [Object Storage for Data Platforms at Scale](#), the field guide to running object storage at scale.

Related posts

- [Object Storage for Data Platforms at Scale](#)
- [Sail: when Apache Spark meets Rust](#)
- [Spark on Kubernetes reserves CPU it never uses](#)
- [Stop using MinIO as a NoSQL database](#)

PDF snapshot of the public article. The online version is authoritative; links remain clickable.