Qumulo & Databricks Technical Integration Report

This comprehensive technical report details the validated integration patterns for co-deploying the Databricks Data Intelligence Platform with the Qumulo Data Ocean. While this guide utilizes Databricks on AWS as a baseline reference for architectural components, the integration principles and configuration parameters apply equally to Databricks deployments on Microsoft Azure and Google Cloud Platform (GCP).

Field	Details
Partner / Platform	Databricks Data Intelligence Platform
Integration Category	Analytics / ML / ETL / Data Engineering
Qumulo Protocol(s)	S3 API (via HTTPS), NFSv4.1, NFSv3, SMB 3.1.1
Qumulo Min Version	7.1.0+ (NFSv4.1 requires 4.3.0+; Kerberos cross-domain trusts require 7.7.5.1+)
Qumulo Version Tested	7.8.0.1 (February 18, 2026; Quarterly Release)
Partner Version(s) Tested	Databricks Runtime 17.3 LTS (Ubuntu 24.04.3 LTS; Apache Spark 4.0.0, Delta Lake 4.0.0, Java Zulu 17.58+21-CA (JDK 17), Scala 2.13.16, Python 3.12.3, R 4.4.2, MLflow: 3.0.1, Databricks Feature Engineering: 0.14.0, Databricks SDK for Python: 0.96.0); All-Purpose Compute clusters (Serverless SQL Warehouses for BI/SQL query workloads)
Overall Status	Validated

1. Architectural Foundation

1.1 Databricks Platform Architecture

To effectively integrate Qumulo, it is crucial to understand the Databricks architecture, which operates on a strict separation of planes:

- **Control Plane:** Managed entirely by Databricks, this backend layer hosts the Workspace UI, notebook management, job scheduling, and the Unity Catalog governance layer.
- **Compute (Data) Plane:** This is where data processing physically occurs. Standard All-Purpose Compute and Job clusters run within your own cloud account's Virtual Private Cloud (VPC) on cloud-native instances (e.g., AWS EC2). Serverless SQL Warehouses run within a secure, Databricks-managed VPC.
- **Storage Layer:** While Databricks requires a small root bucket for workspace configuration, your primary business data resides in your own managed storage. This is where the Qumulo Data Ocean integrates natively, serving as the backend storage tier without requiring data to be migrated into hyperscaler object stores.

1.2 The Medallion Data Architecture

Databricks strongly recommends organizing data processing into the Medallion architecture pattern, utilizing Delta Lake to ensure ACID guarantees:

- **Bronze Layer:** The raw ingestion point for unstructured data, telemetry, and change data capture (CDC) feeds directly from source systems.
- **Silver Layer:** Filtered, cleaned, and augmented data ready for enterprise analytics, data science, and AI training.
- **Gold Layer:** Highly refined, business-level aggregates used for Business Intelligence (BI) and reporting.

Qumulo excels as the foundation for the Bronze and Silver tiers, allowing elastic compute to process massive datasets in place.

2. Integration Pattern: Direct Data Access via S3 API (S3A)

The S3 API integration leverages the Apache Hadoop S3A connector to enable Databricks Spark workloads to read and write Parquet, Delta, JSON, and ORC files directly from Qumulo. This pattern is optimal for standard Delta Lake operations and Lakeflow Auto Loader ingestion.

Note: This pattern requires All-Purpose Compute clusters; Serverless SQL Warehouses do not support custom S3A configurations.

Step 2.1: Qumulo-Side S3 Configuration

Authentication relies on Qumulo-generated S3 access key pairs, which map to Active Directory or local authentication identities to preserve POSIX/ACL permissions.

1. **Generate Keys:** Access the Qumulo qq CLI. To create an S3 access key pair for your own user identity, run:

   ```
   qq s3_create_access_key --self
   ```

   To create an S3 access key for a dedicated Databricks service account, supply the identity prefix:

   ```
   qq s3_create_access_key ad:DATABRICKS_SVC_ACCOUNT
   ```

2. **Save Credentials:** Securely record the returned access_key_id and secret_access_key.
3. **SSL Certificates:** Ensure CA-signed SSL certificates are deployed on your Qumulo cluster, as Databricks JVM trust stores will reject self-signed certificates.

Step 2.2: Databricks-Side Credential Management

Never expose Qumulo S3 credentials in plain text within notebooks.

1. Use the Databricks CLI to create an encrypted secret scope:

   ```
   databricks secrets create-scope qumulo-s3
   ```

2. Store the credentials in the scope:

   ```
   databricks secrets put-secret qumulo-s3 access-key-string-value
   "<your-qumulo-access-key>"
   databricks secrets put-secret qumulo-s3 secret-key-string-value
   "<your-qumulo-secret-key>"
   ```

Step 2.3: Databricks Cluster Spark Configuration

Navigate to your All-Purpose Compute cluster's Advanced Options > Spark Config and append the following properties. These settings enforce path-style addressing, disable proprietary ETag validation, and implement multipart upload purging:

```
spark.hadoop.fs.s3a.endpoint
https://<your-qumulo-cluster-ip>:9000
spark.hadoop.fs.s3a.access.key {{secrets/qumulo-s3/access-key}}
spark.hadoop.fs.s3a.secret.key {{secrets/qumulo-s3/secret-key}}
spark.hadoop.fs.s3a.path.style.access true
spark.hadoop.fs.s3a.signing-algorithm
AWS4UnsignedPayloadSignerType
spark.hadoop.fs.s3a.aws.credentials.provider
org.apache.hadoop.fs.s3a.SimpleAWSCredentialsProvider
spark.hadoop.fs.s3a.endpoint.region us-east-1
spark.hadoop.fs.s3a.region us-east-1
spark.hadoop.fs.s3a.change.detection.mode none
spark.hadoop.fs.s3a.expect.continue false
spark.hadoop.fs.s3a.checksum.enabled false
spark.hadoop.fs.s3a.payload.signing.enabled false
spark.hadoop.fs.s3a.impl org.apache.hadoop.fs.s3a.S3AFileSystem
spark.hadoop.fs.s3a.bucket.probe 0
spark.hadoop.fs.s3a.committer.name directory
spark.databricks.unity.filesystem.s3a.enabled false
spark.databricks.unity.filesystem.s3.enabled false
```

Crucial Configurations Explained:

- **ETag Mismatch:** Setting `fs.s3a.change.detection.mode=none` prevents validation failures, as Qumulo utilizes proprietary ETags rather than AWS MD5 checksums.
- **Multipart Purging:** Qumulo can't auto-clean orphaned MPUs, so you must clean them yourself; here is the safe way. Setting `fs.s3a.directory.operations.purge.uploads = true` which scopes purges to rename/delete operations on a directory rather than the whole bucket, or to schedule `hadoop s3guard uploads -abort -force s3a://bucket/` as a separate housekeeping job.

3. Integration Pattern: POSIX File Access via NFSv4.1

For POSIX-dependent pipelines (e.g., geospatial analysis, bioinformatics, computer vision) and high-throughput MLflow artifact checkpointing, Databricks clusters can mount the Qumulo file system directly via NFSv4.1 using cluster-scoped initialization scripts.

Step 3.1: Qumulo-Side NFS Configuration

1. **Create Export:** Using the Qumulo qq CLI, add a new export:
   ```
   qq nfs_add_export --export-path /databricks_mount --fs-path /databricks_data
   --create-fs-path
   ```
2. **Network Load Balancing:** A single Qumulo node enforces a strict limit of 1,000 concurrent network connections. To prevent connection exhaustion at scale, you must configure Qumulo Floating IPs and Qumulo Authoritative DNS (QDNS). QDNS automatically resolves DNS requests by distributing the floating IPs across the multi-node Qumulo cluster chassis via round-robin, ensuring Databricks connections are evenly load-balanced.

Step 3.2: Databricks Init Script Deployment

Because cloned Databricks worker instances can share hostnames, state collisions may occur over NFSv4.1. Your cluster-scoped init script must generate a unique co_ownerid identifier per node using utilities like uuidgen. The mount command must also specify nconnect=16 to multiplex traffic and optimize throughput for write-heavy workloads.

```
#!/bin/bash
# Generate unique co_ownerid to prevent NFS state collisions between cloned workers
UNIQUE_ID=$(uuidgen)
echo "Setting unique NFS owner ID: $UNIQUE_ID"
# Mount Qumulo directory via NFSv4.1 utilizing the Qumulo DNS round-robin address
mkdir -p /mnt/qumulo_data
mount -t nfs4 -o nconnect=16,rw,hard <qumulo-round-robin-dns>:/databricks_mount /
mnt/qumulo_data
```

4. Integration Pattern: Hive Metastore Federation to Unity Catalog

Databricks Unity Catalog (UC) is designed to work only with native object storage from each cloud it supports, such as S3 for AWS and Blob for Azure. Even though Qumulo runs on object storage for the persistent store, it presents multiple protocols including S3 and NFS, but is not the native object store. Therefore, you cannot register Qumulo directly as a UC-managed external storage location.

To achieve governance on data stored and managed on the Qumulo data platform, the validated pattern is to register Qumulo-backed datasets in the legacy Databricks Hive Metastore, and then use Catalog Federation to mirror the Hive metastore into Unity Catalog as a "foreign catalog".

Step 4.1: Register External Delta Tables in the Hive Metastore

Ensure your cluster is configured with the S3A properties (Pattern 1). Then, create external tables in the legacy hive_metastore catalog that explicitly reference the Qumulo S3 path. The Hive Metastore will track table definitions, while the data physically resides on Qumulo.

```
-- Create database in legacy Hive Metastore
CREATE DATABASE IF NOT EXISTS hive_metastore.qumulo_data;

-- Register external Delta table backed by Qumulo
CREATE TABLE hive_metastore.qumulo_data.sensor_readings
    ( device_id STRING,
    reading_ts TIMESTAMP,
    value DOUBLE
)
USING DELTA
LOCATION 's3a://qumulo-datalake/bronze/sensor_readings';
```

Step 4.2: Federate into Unity Catalog

The hive_metastore is automatically visible within Unity Catalog as a federated catalog, enabling your organization to work with Hive metastore tables in Unity Catalog. When you query these foreign tables, Unity Catalog provides the governance layer (access control checks, auditing, lineage inference), while the query itself runs against the underlying Hive metastore to leverage the latest partition metadata.

Access to foreign tables using Unity Catalog is read-and-write for internal legacy Databricks Hive metastores. New tables and updates committed from the foreign catalog are written back to the Hive metastore, maintaining full interoperability.

Step 4.3: Promote to a UC-Managed Gold Table

While federation provides governance visibility over foreign tables, Databricks recommends migrating datasets that drive production workloads or are frequently queried to Unity Catalog managed tables for the best performance and built-in optimizations.

```
-- Promote Qumulo-backed table to a UC-managed Gold table in cloud storage
CREATE OR REPLACE TABLE qumulo_catalog.production.sensor_data_governed
AS SELECT * FROM hive_metastore.qumulo_data.sensor_readings;
```

Step 4.4: Execute Bi-Directional Synchronization

Because operations like UPDATE or DELETE on the Gold table remain in cloud storage, curated results must be synchronized back to Qumulo to maintain a single source of truth for downstream, non-Databricks applications.

```
-- Sync curated changes back to the Qumulo S3 source of truth
INSERT OVERWRITE hive_metastore.qumulo_data.sensor_readings
SELECT * FROM qumulo_catalog.production.sensor_data_governed;
```

5. Integration Pattern: Delta Sharing Integration

Databricks Delta Sharing ships in three flavors: Databricks-to-Databricks (UC required on both ends), Databricks-managed open sharing (UC still required on the provider side), and a customer-managed deployment of the open-source Delta Sharing reference server. The first two protocols both rely on the Delta Sharing server that is built into Databricks and require Unity Catalog on the provider workspace, so they are unavailable with Qumulo as of this validation. The third path is a customer-managed deployment of the open-source Delta Sharing server, which lets you share from any platform to any platform, whether Databricks or not, and is the only viable route when the backing store is a Qumulo S3A bucket that cannot be promoted to a UC external location.

The architecture we are building therefore looks like this: Delta tables live on a Qumulo cluster, exposed via its native S3 API; an open-source Delta Sharing server (a small JVM service we host ourselves) sits in front of those tables, authenticates recipients with a bearer token, and hands out short-lived pre-signed URLs that point back to the Qumulo S3 endpoint; Databricks workspaces (or any other Delta Sharing client — pandas, Spark, Power BI, Trino) consume the share through a .share profile file. The Delta Sharing server is on the control path only where actual Parquet bytes flow directly from Qumulo to the recipient.

Prerequisites

On the Qumulo side, the S3 API service must be enabled on the cluster, you must have an installed SSL certificate (the cluster's self-signed cert works for prototyping but several Java HTTP clients will reject it),

and an identity with `PRIVILEGE_S3_BUCKETS_WRITE` plus directory-create permission under the bucket root. Note two Qumulo-specific constraints that affect the rest of the implementation details: Qumulo Core supports only AWS Signature Version 4 (SigV4) authentication and only path-style bucket addressing. Hadoop's S3A client supports both.

On the sharing-server side, plan for a small Linux VM (or container) with Java 8/11, network reachability to the Qumulo cluster on port 9000, and a public-or-routable TCP port for the Delta Sharing REST endpoint (default 8080). On the Databricks side, no UC is required for consumption, the workspace just needs the delta-sharing-spark library (bundled in modern DBRs) and network egress to both the sharing server and the Qumulo S3 endpoint where the pre-signed URLs resolve.

Step 1 — Prepare the Qumulo S3A bucket and credentials

Create the bucket and write a Delta table into it. Using the qq CLI:

```shell
# Create the bucket
qq s3_add_bucket --name analytics-shared --create-fs-path \
  --fs-path /shared/analytics

# Mint an S3 access key pair for the service account
qq s3_create_access_key --user share-svc
# → returns access_key_id and secret_access_key (save the secret now;
#   Qumulo never re-displays it)
```

Apply an inheritable ACL on /shared/analytics granting read for share-svc so the access key can list and GET every Parquet file and _delta_log entry the Sharing server will reference. Confirm with the AWS CLI pointed at the Qumulo endpoint:

```shell
aws --endpoint-url https://qumulo.internal:9000 \
    --profile share-svc \
    s3 ls s3://analytics-shared/sales_facts/_delta_log/
```

Write the Delta table itself from any Spark job (Databricks, an EMR cluster, a local Spark shell) configured with `spark.hadoop.fs.s3a.endpoint=https://qumulo.internal:9000`, `fs.s3a.path.style.access=true`, and the `share-svc` keys. From here on, Qumulo is the durable store.

Step 2 — Stand up the open-source Delta Sharing server

Pull the reference server release (the project is the same delta-io/delta-sharing repo that ships the protocol spec and clients) and either run the binary directly or use the published Docker image. The server's behavior is driven entirely by a YAML config file.

Step 3 — Point the YAML at Qumulo

This is the step that diverges from the canonical AWS example, because the server defaults to talking to s3.amazonaws.com. The template ships with s3a:// examples, but on its own that scheme will resolve against real AWS — issue #753 in the upstream repo documents exactly this failure mode for users on Dell ECS, and the fix applies verbatim to Qumulo: you must inject Hadoop S3A overrides so the embedded client targets the Qumulo endpoint with path-style addressing.

A working delta-sharing-server.yaml for Qumulo:

None

```
version: 1
shares:
  - name: "qumulo-share"
    schemas:
      - name: "analytics"
        tables:
          - name: "sales_facts"
            location: "s3a://analytics-shared/sales_facts"
            id: "11111111-1111-1111-1111-111111111111"

# Qumulo S3 endpoint overrides — these are read by the embedded
# Hadoop S3A FileSystem the server uses to list & presign objects.
# Without them the server will try sales_facts.s3.amazonaws.com.
storage:
  type: s3a
  properties:
    fs.s3a.endpoint: "https://qumulo.internal:9000"
    fs.s3a.access.key: "AKIA...QUMULO_KEY"
    fs.s3a.secret.key: "...QUMULO_SECRET..."
    fs.s3a.path.style.access: "true"
    fs.s3a.connection.ssl.enabled: "true"
    fs.s3a.aws.credentials.provider:
"org.apache.hadoop.fs.s3a.SimpleAWSCredentialsProvider"
    fs.s3a.signing-algorithm: "AWS4UnsignedPayloadSignerType"

host: "0.0.0.0"
port: 8080
endpoint: "/delta-sharing"
preSignedUrlTimeoutSeconds: 3600
deltaTableCacheSize: 500
authorization:
  bearerToken: "REPLACE_WITH_LONG_RANDOM_TOKEN"
```

A few choices worth justifying explicitly in the report:

- `fs.s3a.path.style.access: "true"` is non-negotiable. Virtual-hosted style URLs (`bucket.qumulo.internal`) will fail because Qumulo only accepts path-style.
- `AWS4UnsignedPayloadSignerType` matches Qumulo's SigV4-only enforcement; leaving the default in older S3A versions could allow SigV2 attempts that get rejected with `AuthorizationHeaderMalformed`.
- `preSignedUrlTimeoutSeconds: 3600` is the validity window of the URLs that clients receive. Match it to the longest scan you expect from a recipient, but no longer — these URLs grant direct read on Qumulo objects.
- Running on `host: 0.0.0.0` in this example; in production terminate TLS at a reverse proxy (nginx, Envoy) in front of the server and bind the server itself to localhost.

Start the server (Docker variant):

Shell

```
docker run -d --name delta-sharing-server \
  -p 8080:8080 \
  -v /etc/delta-sharing/conf.yaml:/conf/delta-sharing-server.yaml \
  deltaio/delta-sharing-server:latest \
  --config /conf/delta-sharing-server.yaml
```

Smoke-test from the host with curl: `curl -H "Authorization: Bearer <token>" http://localhost:8080/delta-sharing/shares` should return `qumulo-share`.

Step 4 — Consume from Databricks

Distribute a .share profile file to each recipient. The file is a small JSON document; no Unity Catalog object is created anywhere:

```json
{
  "shareCredentialsVersion": 1,
  "endpoint": "https://sharing.yourcompany.com/delta-sharing",
  "bearerToken": "REPLACE_WITH_LONG_RANDOM_TOKEN",
  "expirationTime": "2027-01-01T00:00:00.0Z"
}
```

In a Databricks notebook (any workspace, UC-enabled or not):

```python
import delta_sharing

profile = "/dbfs/FileStore/qumulo.share"
table_url = f"{profile}#qumulo-share.analytics.sales_facts"

df = delta_sharing.load_as_spark(table_url)
df.createOrReplaceTempView("sales_facts")
spark.sql("SELECT region, SUM(amount) FROM sales_facts GROUP BY region").show()
```

When load_as_spark is invoked, the Spark driver hits the sharing server, receives a list of pre-signed Qumulo URLs (signed by the server's S3A credentials, valid for one hour), and then Spark executors fetch the Parquet directly from https://qumulo.internal:9000/analytics-shared/.... The sharing server never sees the data bytes — it only sees metadata queries and authorization checks.

Operational considerations

Network reachability. This is the most common production failure. The pre-signed URLs the server hands out point at the Qumulo endpoint *as the server sees it*. If the server uses https://qumulo.internal:9000 and the Databricks workspace cannot resolve or reach that hostname, every shared read fails with a connection error even though the metadata calls succeed. Either publish Qumulo behind a reachable DNS name (and matching cert), or run the sharing server in a network position where the Qumulo URL it generates is also valid for the recipient. For cloud Databricks recipients, this almost always means PrivateLink, an inbound VPN, or a reverse proxy that fronts both the sharing endpoint and the Qumulo S3 endpoint.

Authorization model. The reference server's bearer-token auth is a single shared secret per authorization block. For multi-tenant sharing, deploy multiple server instances (or a token-aware proxy) and rotate tokens out-of-band; the OSS server does not implement per-recipient ACLs the way UC-managed sharing does.

Concurrent writers. Delta on S3-compatible storage without a coordinated commit service is single-writer-safe only. Confine writes into the Qumulo bucket to a single Spark driver, and let the Sharing server stay read-only.

Feature gap vs. Databricks-managed sharing. Customer-managed open sharing covers tables and CDF; it does not cover notebook sharing, volume sharing, AI model sharing, or UC-level audit and usage tracking. Those features are exclusive to the Databricks-managed protocols and should be called out as scope exclusions in the report's "limitations" section.

Certificates. The Hadoop S3A client used inside the server validates the Qumulo TLS cert by default. Either install a CA-signed cert on Qumulo or import the cluster cert into the server JVM's truststore — do not weaken fs.s3a.connection.ssl.enabled to false for production.

Qumulo provides the durable Delta storage via its S3A API; the open-source Delta Sharing server bridges Qumulo into the Delta Sharing protocol that Databricks natively understands; and Unity Catalog never enters the picture on either side.