Liquibase

CHEAT SHEET

Liquibase Secure + Databricks Schema Change Governance

Built for data engineers running Unity Catalog, Delta tables, and automated releases. Keep this open while you wire up your first governed pipeline.

databricks

Introduction

Databricks gives you governance for access and lineage. Delta gives you table history and recovery. The missing piece is governance for schema change itself.

Liquibase Secure fills that gap with a controlled path for schema change that is **reviewable**, **repeatable**, **policy enforced before deploy**, and **auditable after deploy**.

Who this is for

If you ship schema changes into Databricks SQL warehouses or jobs, and you care about not breaking downstream models, dashboards, pipelines, and feature inputs, this is for you.

What this cheat sheet covers

1. Liquibase Secure first workflow
2. Connection patterns
3. Governance controls
4. DDL gotchas
5. Recovery patterns

A note on terminology: *"Warehouse"* and *"SQL Warehouse"* refer to Databricks SQL Warehouses, which are compute resources that let you run SQL commands on data in Databricks. They were formerly known as Databricks SQL Endpoints.

Where Liquibase Community fits

Liquibase Community is a free open-source tool that helps track, version, and deploy database schema changes. Liquibase Secure is a commercial offering that adds enhanced governance, auditing, security, and enterprise features on top of the open-source core.

The Databricks native workflow, completed with Liquibase Secure

The principle

Databricks already gives you governance and history. Unity Catalog provides centralized access control, auditing, and lineage across workspaces.

Delta provides table versions and supports restoring a table to an earlier version or timestamp.

The gap is still the same:

Schema change needs to be **reviewable**, **repeatable**, and **constrained before it lands** in shared, business critical environments and starts feeding scheduled jobs, dashboards, and model features.

⚠ One bad schema change can cascade into failed jobs, broken dashboards, and incorrect model inputs.

Reference workflow

01

Developer opens a PR that includes a changelog update

02

CI runs liquibase validate to confirm the changelog parses correctly

03

CI runs Liquibase Secure policy checks to block risky patterns before deploy

04

CD deploys via Liquibase Flows to a staging catalog or schema

05

CD deploys the same changelog to prod with approvals and the same pipeline identity

06

Liquibase Secure generates operation reports that provide what changes were made, when they ran, and where they were deployed

07

Liquibase Secure drift visibility detects out of process schema changes after deployment

Deployment identity rule: Run deployments with a service principal, not a human user. The deployment identity should be the standard user id for schema change, so changes are controlled and recorded.

Databricks JDBC connection, authentication, and authorization

Get the JDBC URL the right way

For a Databricks SQL warehouse, you can copy the JDBC URL from the warehouse connection details, or construct it from server hostname, port, and HTTP path.

Minimum JDBC shape:

```
jdbc:databricks://<server-hostname>:443;httpPath=<http-path>;AuthMech=<auth-mech>;...
```

Databricks forms the HTTP address by appending the httpPath value to host and port.

Authentication choices with the right posture

Personal access tokens exist, but Databricks positions them as best suited for testing. For shared environments, align to OAuth where possible.

A common PAT based JDBC pattern looks like:

```
jdbc:databricks://:443;httpPath=;AuthMech=3;UID=token;PWD=
```

For shared environments, align to OAuth where possible. The Databricks JDBC driver supports OAuth tokens for users and service principals.

Liquibase Secure simplifies authentication setup with specific properties that handle authentication mechanisms without manually constructing JDBC URLs. This reduces configuration errors and makes it easier to implement the right security posture across environments.

Authorization in Unity Catalog

Authentication gets you connected. Authorization in Unity Catalog determines what your deployment identity can change.

Recommended posture:

- Use a service principal as the deployment identity
- Grant only the minimum permissions needed to apply approved schema changes
- Reduce or remove human privileges to change object structures directly in shared environments

This is how you keep interactive users out of high risk write privileges while still shipping safely.

Liquibase Secure plus Databricks

Liquibase Secure supports Databricks SQL warehouses with Unity Catalog aware workflows, environment specific configuration patterns, and governed deployments across dev, staging, and prod.

Databricks native operations like time travel and CLONE are often used in testing and recovery workflows alongside Liquibase Secure governed deployments.

Some advanced Databricks operations depend on Unity Catalog system tables. If those are restricted, plan accordingly.

The 5 essential Liquibase commands for governed Databricks pipelines

These are the core Liquibase CLI verbs used in a governed Databricks deployment flow. **Liquibase Secure is the recommended distribution** for shared, business critical Unity Catalog environments because it adds policy enforcement, drift visibility, and audit ready evidence.

ⓘ Liquibase Flows is the recommended way to standardize these steps across pipelines and environments.

validate

Command: `liquibase validate`

Purpose: Confirm the changelog parses cleanly in CI before you touch a shared environment.

checks run

Command: `liquibase checks run`

Purpose: Fail fast on risky patterns, forbidden operations, or missing required metadata.

update

Command: `liquibase update`

Purpose: Apply the approved change set to the target catalog or schema in a repeatable way.

history

Command: `liquibase history`

Purpose: See what ran, when it ran, and which change sets were applied without stitching together logs and system history from multiple sources.

rollback

Command: `liquibase rollback`

Purpose: Reverse schema cleanly, rather than relying on emergency edits in a notebook or some other manual manipulation of the infrastructure.

Unity Catalog environment mapping that works in practice

Unity Catalog is the governance plane across workspaces, including access control, auditing, and lineage.

A simple mapping pattern:

dev catalog	staging catalog	prod catalog

Build once, deploy many:

Keep the changelog identical across all environments. The same tested changeset that runs in dev is what deploys to staging and prod. Only environment-specific properties (connection details, catalog names, credentials) change between deployments. This ensures what you test is exactly what you deploy, eliminating environment-specific variations that cause surprises in production.

Practical note for data platforms:

Guardrails matter most where schema changes impact shared datasets across Bronze, Silver, and Gold tiers and the scheduled jobs and consumers that depend on them.

Liquibase Secure controls that prevent outages

1

Layer 1: Policy checks before deployment

Policy checks block unsafe patterns before they land.

Common examples for data teams:

- block drops and renames on protected tables
- enforce naming conventions
- require ownership metadata
- require labels or tags for sensitive datasets
- prevent changes that violate internal data contracts

2

Layer 2: Drift visibility after deployment

Policy checks stop bad changes in process. **Drift visibility catches changes made out of process** whether they are benign or malignant in their intent.

This is also how you prove whether a schema change came through the governed path or not.

3

Layer 3: Evidence by default

Your goal is audit ready traceability without manual prep.

Capture:

- What ran
- Where it ran
- Who approved it
- The exact changelog version tied to the commit
- The pipeline run ID

Liquibase Secure is designed to produce these receipts so you are not stitching evidence together from notebooks and job logs.

Databricks DDL gotchas you will hit

1. Drop column and rename column have dependencies

When you drop a column or field, you must drop dependent check constraints and generated columns.

When you rename a column, you must update dependent check constraints and generated columns. Primary keys and foreign keys using the column will be dropped.

Translation for engineers:

Plan dependency cleanup as part of the same governed change set. Otherwise your simple schema edit becomes a release breaker.

2. Type changes can break downstream behavior

Type changes can have downstream effects. Even widening numeric precision can change aggregations, casts, and BI behavior.

Note: Treat type changes as contract changes, not a casual refactor.

3. Constraints behave differently than classic OLTP expectations

Databricks verifies existing rows satisfy a CHECK constraint before adding it.

That verification can be expensive. Treat it like a migration with runtime and cost impact, not a free DDL tweak.

Note: Foreign keys and primary keys are informational and supported only for Unity Catalog tables, not hive_metastore.

4. Clone strategy has sharp limits with security policies

If you use row filters or column masks, deep and shallow clones are not supported on those tables.

Note: If your rollout plan depends on cloning prod data into a test environment, you need a fallback for governed datasets that use masking.

Recovery and rollback without chaos

Use Delta time travel and RESTORE for table state recovery. RESTORE is great when:

data state needs to be reverted	a bad write landed and you need to reset the table content

Use controlled rollbacks for schema integrity.

Delta RESTORE is not a substitute for disciplined schema rollback.

Schema rollback is about reversing the planned structural change, in the right order, with dependencies accounted for.

The goal is to avoid someone hotfixing a shared environment in a notebook becoming your normal incident response.

CI and CD implementation patterns that pass a security review

Rule: No human passwords in pipelines.

Recommended posture

- Run the pipeline job as a service principal
- Use OAuth where possible
- If you must use a personal access token, treat it as transitional, rotate it, and scope it tightly

Reduce or remove human privileges to change structures directly in shared environments. The automation identity should be the primary path for schema change.	Run Liquibase Secure as the deployment step for schema changes under that pipeline identity, so every change is executed through the governed path and recorded.

Quick Checklist

Phase	Checklist Item
Before you onboard	☐ Unity Catalog is the governance plane you will deploy into
	☐ The deployment identity is a service principal
	☐ You can source server hostname and httpPath from warehouse connection details
	☐ Confirm the service principal has only the permissions needed for approved schema changes
Before each release	☐ Run liquibase validate in CI
	☐ Run liquibase checks run to enforce policy gates (Liquibase Secure)
	☐ Deploy via standardized pipeline steps using Liquibase Flows, using a service principal
	☐ CD deploys the same changelog to prod with approvals
	☐ Capture commit hash, pipeline run ID, and Secure evidence receipts tied to the changelog version
Before risky schema operations	☐ Check dependencies for drops and renames
	☐ Treat type changes as contract changes and validate downstream behavior
	☐ Plan CHECK constraint verification cost
For incident recovery	☐ Use Delta time travel and RESTORE for table state rollback
	☐ Use controlled rollbacks for schema intent

Ready to govern Databricks schema changes without slowing delivery?

Liquibase Secure helps you enforce standards, block risky changes before deploy, detect drift after deploy, and keep an audit ready change trail tied to every run.

Request a Live Demo

About Liquibase

Liquibase empowers organizations to deliver mission-critical applications, data products, and AI initiatives by automating and governing database change.

Liquibase Secure is the enterprise platform for managing database change with velocity, security, and control. It brings database delivery into modern development workflows by automating deployments, enforcing policies before changes reach production, detecting unauthorized drift, supporting recovery, and maintaining detailed audit evidence across 70+ database platforms.

Developers and platform teams gain a standardized, self-service path for delivering database changes without waiting on tickets, manual reviews, or one-off scripts. Security and compliance teams gain consistent guardrails, separation of duties, traceability, and evidence across database environments. The result is faster delivery without sacrificing reliability, security, or compliance.

Liquibase is also the company behind Liquibase Community, a project with deep open-source roots that has been downloaded more than 100 million times and is trusted by thousands of teams worldwide. Liquibase Secure builds on that foundation to provide the automation, governance, and enterprise support required for mission-critical database change at scale.

© 2026 Liquibase Inc. All Rights Reserved