**Databricks Unity Catalog Technical Design Document**

**Use case:**

This Databricks Unity Catalog design establishes a secure, scalable, and region-aware data governance framework for an organization with Sales, Marketing, and eCommerce lines of business, supporting approximately 3,000 users distributed across multiple regions.

Each region will have dedicated metastores to ensure data residency compliance, with logically separated catalogs (sales\_data, marketing data, ecommerce\_data) containing standardized schemas for raw, curated, and analytics data.

Access control is managed centrally through identity federation (e.g., Azure AD) and SCIM-provisioned groups mapped to clearly defined roles with least-privilege permissions.

Infrastructure as Code (Terraform) will automate creation and management of metastores, catalogs, and policies, while centralized audit logging and monitoring ensure compliance and traceability.

This setup provides a consistent, secure foundation for collaborative data management and analytics across all business units.

**0. Design Objectives**

**Why we’re doing this:**

* Establish a unified data governance layer across all business domains.
* Simplify secure data sharing between Sales, Marketing, and eCommerce.
* Ensure region-specific compliance (e.g., GDPR).
* Centralize lineage, auditing, and discoverability.
* Scale to 3,000+ users with role-based access.

**1. Governance Model and Guiding Principles**

**Separation of Concerns**

* Each line of business (LoB) has logically isolated catalogs.
* Shared data lives in dedicated "Cross-LoB" catalogs.

**Least Privilege Access**

* Users and service principals only access what they require.

**Central Control**

* Unity Catalog Metastore acts as the single source of truth.

**Multi-Region Strategy**

* Dedicated metastores per region for compliance.
* Common governance framework across metastores.

**Automated Entitlements**

* All permissions managed declaratively (Infrastructure as Code).

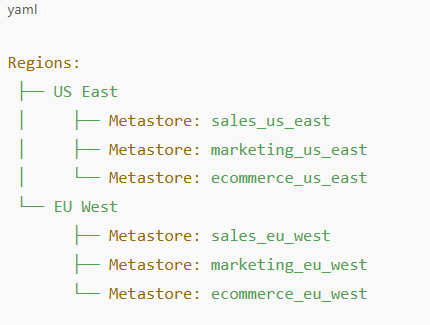
**2. Metastore Architecture**

**Best Practice:** A Unity Catalog *Metastore* is region-specific.

**Proposed Setup:**

* **Metastore Naming Convention:**
  + metastore\_sales\_<region>
  + metastore\_marketing\_<region>
  + metastore\_ecommerce\_<region>
* **Region Example:**
  + metastore\_sales\_us\_east
  + metastore\_sales\_eu\_west
  + etc.
* **Shared Data:**
  + Optional “Global Shared Metastore” if cross-region replication is needed (advanced).

**Diagram (Textual):**

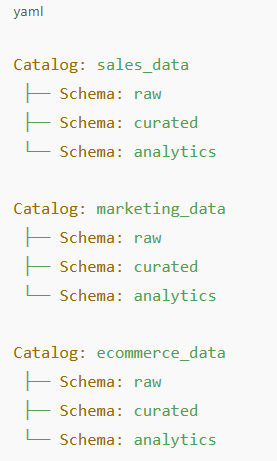
****

**Best Practice:**

Start with one Metastore per region and expand as necessary. Avoid a single Metastore spanning multiple regions due to latency and compliance.

**3. Catalog and Schema Design**

**Logical Catalog Structure per Metastore:**



**Special Catalogs:**

* shared\_data: Cross-LoB data accessible to all business units.
* sandbox: User experimentation (ephemeral).

**Comments:**

* **Raw:** Direct ingestion (e.g., Delta Live Tables).
* **Curated:** Cleaned, validated data.
* **Analytics:** BI-ready datasets.

**Best Practice:**

Use clear naming conventions (<lineofbusiness>\_data) and avoid ambiguous names like default.

**4. Access Control Design**

**Approach:**

* Use **Unity Catalog GRANT statements** to define permissions.
* Manage access through **Groups**, mapped from your Identity Provider (Azure AD/Okta).

**Groups Example:**

python-repl

CopyEdit

sales\_readers

sales\_writers

sales\_admins

marketing\_readers

...

**Role Mapping:**

|  |  |
| --- | --- |
| Role | Permissions |
| Readers | SELECT |
| Writers | SELECT, INSERT, UPDATE |
| Admins | ALL PRIVILEGES + OWNERSHIP |
| Data Stewards | ALL on curated + analytics schemas |
| Data Engineers | ALL on raw + curated schemas |

**Sample GRANT statements:**

sql

CopyEdit

GRANT USAGE ON CATALOG sales\_data TO `sales\_readers`;

GRANT SELECT ON SCHEMA sales\_data.analytics TO `sales\_readers`;

GRANT ALL PRIVILEGES ON SCHEMA sales\_data.curated TO `sales\_writers`;

**Best Practice:**

Avoid granting permissions to individual users—use groups consistently.

**5. Identity Federation & Authentication**

**Options:**

* Azure Active Directory integration (recommended).
* SCIM provisioning for automatic user and group sync.

**Setup Steps:**

1. Integrate Databricks workspace with AAD tenant.
2. Enable SCIM connector.
3. Map AAD groups to Databricks groups.
4. Automate group membership management.

**Best Practice:**

Use service principals for automation pipelines rather than user tokens.

**6. Region Strategy and Data Residency**

**Requirements:**

* Keep EU data within EU regions.
* Keep US data within US regions.
* Minimize cross-region traffic.

**Design Notes:**

* Each region’s Metastore manages its own S3/ADLS storage accounts.
* Region-specific workspaces connect to the matching Metastore.
* Optional: Use Delta Sharing for cross-region sharing without copying.

**7. Automation and CI/CD**

**Infrastructure as Code (IaC):**

* **Tools:**
  + Terraform (preferred)
  + Databricks CLI
* **Modules to create:**
  + Metastore resources
  + Catalogs and schemas
  + Access control policies
  + Storage credentials
  + External locations

**Example Terraform Module:**

hcl

CopyEdit

module "sales\_metastore" {

source = "./modules/unity\_catalog\_metastore"

region = "us-east"

catalog\_names = ["sales\_data"]

schemas = ["raw", "curated", "analytics"]

group\_policies = var.sales\_group\_policies

}

**Best Practice:**

All grants and resources declared in code—never manually created in the console.

**8. Monitoring and Auditing**

**Tools:**

* Unity Catalog audit logs (accessible via the account console).
* Cluster-level audit logs.
* Event Hub / Event Grid (if on Azure).
* Alerts for:
  + Unauthorized access attempts.
  + High-volume exports.

**Best Practice:**

Centralize audit logs in a dedicated logging workspace or SIEM.

**9. Deployment Plan**

**Identity & Access:**

* Integrate AAD/Okta.
* Sync groups via SCIM.

**Storage:**

* Create region-specific storage accounts/buckets.
* Define storage credentials.

**Metastore:**

* Deploy Metastore per region.
* Assign to target workspaces.

**Catalogs and Schemas:**

* Create catalogs (sales\_data, marketing\_data, etc.).
* Create schemas (raw, curated, analytics).

**Access Policies:**

* Grant privileges to groups.

**Automation:**

* Write and test Terraform modules.
* Pipeline deployment.

**Testing:**

* Validate permissions.
* Validate data residency.

**Training:**

* Educate users on naming conventions and access.

**Go-Live:**

* Enable Unity Catalog in production workspaces.
* Monitor initial usage.

**10. Best Practices Recap**

* **Metastore per region.**
* **Declarative access management.**
* **Consistent naming.**
* **Use groups, not individuals.**
* **Automate everything.**
* **Keep data in-region.**
* **Regular audits and reviews.**

This **Databricks Unity Catalog design** is recommended for:

**Ideal Audience For:**

* **Large or mid-sized enterprises** with **multiple lines of business** (e.g., Sales, Marketing, eCommerce).
* Organizations with **hundreds to thousands of data consumers and producers** (you mentioned ~3,000 users).
* Companies operating in **multiple regions** where:
  + Data residency regulations (GDPR, HIPAA, etc.) apply.
  + Performance and latency need to be optimized locally.
* Enterprises adopting **Lakehouse architecture** that want:
  + Consistent governance across all data assets.
  + Centralized lineage, auditability, and access controls.
  + Separation of duties between data producers and consumers.
* Teams with **dedicated platform engineering, security, and compliance functions** that can maintain:
  + Infrastructure-as-Code (Terraform) deployment pipelines.
  + Managed identity integrations (AAD or Okta).
  + Strict role-based access control policies.
* Organizations looking to **migrate from legacy data lakes or Hadoop clusters to Databricks** with Unity Catalog as the unified governance layer.

**Typical Stakeholders Who Should Be Involved:**

* **Data Platform Engineering Teams** – to design and deploy the metastore, catalogs, and automation pipelines.
* **Cloud Infrastructure and Security Teams** – to integrate identity providers and storage accounts.
* **Data Governance and Compliance Teams** – to define access policies and ensure regulatory compliance.
* **Business Data Owners** – from Sales, Marketing, and eCommerce to validate catalog structure and naming conventions.

**Final Note:**

This design is recommended for **enterprises with multi-region operations and large, cross-functional data teams** seeking secure, scalable, and auditable data governance through Databricks Unity Catalog

Best Practise #2

|  |
| --- |
|  |
| **🌍 Best Practice: Start with One Metastore per Region and Expand as Necessary—Avoid a Single Metastore Spanning Multiple Regions**  **🎯 Objective**  Deploy metastores in a way that is **scalable, performant, and compliant**, so you can: ✅ Minimize latency for users and workloads. ✅ Meet data residency and regulatory requirements. ✅ Avoid operational complexity as you grow.  **🟢 Why This Matters**  While it can be tempting to deploy a **single global metastore**, this approach creates significant challenges:   * **Increased Latency:** Cross-region catalog operations can become slow and unreliable. * **Compliance Risks:** Many regulations (e.g., GDPR, HIPAA) require data to remain within a specific geography. * **Complex Recovery:** A failure or misconfiguration impacts all regions at once. * **Limited Scalability:** Regional growth often requires distinct configurations.   Deploying **one metastore per region** ensures data stays local, performance is predictable, and compliance boundaries remain clear.  **🟢 Recommended Approach**  ✅ **One Metastore per Region:**   | **Region** | **Metastore Name** | | --- | --- | | East US | metastore\_us\_east | | West Europe | metastore\_eu\_west | | Asia Pacific | metastore\_apac |   ✅ **Assign Only Local Workspaces:** Each workspace in a region links to its dedicated metastore.  ✅ **Expand Gradually:** Start with a single metastore in each region and create additional ones only if:   * You need further isolation (e.g., separate business units). * You have distinct compliance domains (e.g., public vs. restricted data).   **🟢 Example Workflow**  1️⃣ **Deploy a Metastore per Region:** ✅ Terraform Example:  hcl  CopyEdit  resource "databricks\_metastore" "us\_east" {  name = "metastore\_us\_east"  storage\_root = "abfss://unitycatalog@datalakeeast.dfs.core.windows.net/"  region = "eastus"  }  2️⃣ **Assign Workspaces to the Regional Metastore:** ✅ Terraform Example:  hcl  CopyEdit  resource "databricks\_metastore\_assignment" "sales\_us" {  metastore\_id = databricks\_metastore.us\_east.id  workspace\_id = var.sales\_us\_workspace\_id  default\_catalog\_name = "main"  }  3️⃣ **Validate Assignments:** Confirm each workspace uses the correct metastore and can create catalogs.  **🟢 Best Practices for Success**  ✅ **Keep Metastores Region-Scoped:** Never assign a single metastore across multiple continents.  ✅ **Document Each Metastore:** Record:   * Name * Region * Purpose * Assigned workspaces   ✅ **Review Annually:** Assess whether growth or compliance needs require more granular metastores.  ✅ **Plan for Disaster Recovery:** Ensure storage locations are resilient within the same geography.  ✅ **Avoid Over-Partitioning:** Don’t create multiple metastores in the same region without a clear business justification.  **🟢 Sample Communication Announcement**  **Subject:** 🌍 Regional Metastore Deployment Strategy  **Body:**  To optimize performance and ensure compliance, each Databricks region will operate its own dedicated Unity Catalog metastore.  **Key Points:**   * One metastore per region—no cross-region assignments. * Data stays local to meet regulatory requirements. * Additional metastores will be created only when needed for specific isolation.   **Example:**   * East US: metastore\_us\_east * West Europe: metastore\_eu\_west   For help configuring or migrating metastores, contact [Data Platform Engineering]. |

Best Practise #3

|  |
| --- |
|  |
| **🏷️ Best Practice: Use Clear Naming Conventions (<lineofbusiness>\_data) and Avoid Ambiguous Names Like default**  **🎯 Objective**  Adopt a consistent naming strategy across all Unity Catalog resources to: ✅ Make it immediately clear who owns what. ✅ Avoid conflicts between teams or regions. ✅ Simplify automation, governance, and support.  **🟢 Why This Matters**  Ambiguous or generic names like default, main, or project1 create:   * Confusion over ownership and purpose. * Difficulty automating permissions or pipelines. * Risk of accidental misuse or deletion. * Extra effort to audit and maintain over time.   Clear, descriptive names make your environment **self-documenting** and easier to scale.  **🟢 Recommended Approach**  ✅ **Use the <lineofbusiness>\_data Naming Pattern:**   | **Line of Business** | **Catalog Name** | | --- | --- | | Sales | sales\_data | | Marketing | marketing\_data | | eCommerce | ecommerce\_data |   ✅ **Apply Consistent Naming to Schemas:** Use lifecycle-based names within each catalog:   * raw * curated * analytics   ✅ **Avoid Generic Names:** Never use names like:   * default * test * misc * main (unless specifically reserved)   **🟢 Example Workflow**  1️⃣ **Define Naming Standards in a Shared Document:** Document your approved prefixes and examples.  2️⃣ **Create Catalogs with Clear Names:** ✅ SQL Example:  sql  CopyEdit  CREATE CATALOG sales\_data COMMENT "Sales line of business data catalog";  ✅ Terraform Example:  hcl  CopyEdit  resource "databricks\_catalog" "sales\_data" {  name = "sales\_data"  comment = "Sales LOB catalog"  }  3️⃣ **Use Naming Conventions in All Automation:** Configure your Terraform modules and CI/CD pipelines to enforce naming standards.  4️⃣ **Reject Requests for Noncompliant Names:** Review all change requests to ensure consistency.  **🟢 Best Practices for Success**  ✅ **Be Descriptive:** Include purpose and ownership in names (e.g., sales\_data, finance\_reporting).  ✅ **Use Lowercase with Underscores:** Improves readability and avoids case-sensitive issues.  ✅ **Document Everything:** Maintain a catalog registry with:   * Name * Description * Owner * Region * Purpose   ✅ **Review Quarterly:** Audit for any drift from naming conventions.  ✅ **Train Teams:** Ensure everyone understands and follows the standard.  **🟢 Sample Communication Announcement**  **Subject:** 🏷️ Naming Standards for Unity Catalog Resources  **Body:**  To maintain clarity and prevent ambiguity, all Unity Catalog resources must follow our approved naming conventions.  **Effective Immediately:**   * Use <lineofbusiness>\_data format for catalogs. * Use clear schema names (raw, curated, analytics). * Avoid generic names like default or main.   **Example:** ✅ sales\_data.raw ✅ marketing\_data.analytics  For help applying naming conventions or migrating existing resources, contact [Data Platform Governance Team]. |

Best Practise # 4

|  |
| --- |
|  |
| **👥 Best Practice: Avoid Granting Permissions to Individual Users—Use Groups Consistently**  **🎯 Objective**  Ensure all Unity Catalog and workspace permissions are assigned **exclusively through groups**, so you can: ✅ Simplify permission management as teams grow. ✅ Reduce the risk of hidden, one-off access grants. ✅ Maintain clear, auditable records of entitlements.  **🟢 Why This Matters**  Granting permissions directly to individual users creates:   * Inconsistent access across environments. * Manual work to review and revoke entitlements. * Compliance risks when employees transfer or leave. * Drift between your intended security model and reality.   Groups provide **predictable, role-based access** that scales with your organization.  **🟢 Recommended Approach**  ✅ **Establish Role-Based Groups for Each Line of Business:**   |  |  |  |  | | --- | --- | --- | --- | | **Line of Business** | **Read-Only Group** | **Read/Write Group** | **Admin Group** | | Sales | sales\_readers | sales\_writers | sales\_admins | | Marketing | marketing\_readers | marketing\_writers | marketing\_admins | | eCommerce | ecommerce\_readers | ecommerce\_writers | ecommerce\_admins |   ✅ **Always grant permissions to these groups—never to a person’s username.**  ✅ **Integrate Groups via SCIM:** Automatically sync them from Azure AD or Okta so membership stays up to date.  **🟢 Example Workflow**  1️⃣ **Create Groups in Your Identity Provider:**   * E.g., Azure AD security groups or Okta groups.   2️⃣ **Sync Groups into Databricks via SCIM:**   * Verify group membership aligns with business roles.   3️⃣ **Grant Permissions to Groups Only:**  ✅ **SQL Example:**  sql  CopyEdit  -- Grant usage on catalog  GRANT USAGE ON CATALOG sales\_data TO `sales\_readers`;  -- Grant select on all tables  GRANT SELECT ON ALL TABLES IN SCHEMA sales\_data.analytics TO `sales\_readers`;  -- Grant read/write permissions  GRANT INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA sales\_data.analytics TO `sales\_writers`;  ✅ **Terraform Example:**  hcl  CopyEdit  resource "databricks\_grants" "sales\_analytics" {  schema = "sales\_data.analytics"  grant {  principal = "sales\_readers"  privileges = ["SELECT"]  }  grant {  principal = "sales\_writers"  privileges = ["SELECT", "INSERT", "UPDATE", "DELETE"]  }  }  **🟢 Best Practices for Success**  ✅ **Never Grant to Individuals:** Remove any existing direct grants to user emails.  ✅ **Enforce Group Usage in Policy:** Require all access requests to specify a target group.  ✅ **Review Group Membership Quarterly:** Confirm only current employees have access.  ✅ **Document Group Purpose:** Keep a clear description of each group’s role and scope.  ✅ **Audit Permissions Regularly:** Verify no accidental individual grants were created.  **🟢 Sample Communication Announcement**  **Subject:** 👥 Access Policy Update—Groups Required for All Permissions  **Body:**  To improve security and compliance, all Databricks access must now be managed exclusively through groups.  **Effective Immediately:**   * No permissions will be granted directly to individual users. * All access requests must specify an approved group. * Any existing individual grants will be migrated to group-based assignments.   **Why This Change?**   * Simplifies onboarding and offboarding. * Improves visibility and auditing. * Aligns with our least-privilege security model.   For help mapping users to the appropriate groups, contact [Data Platform Team]. |

Best Practise # 5

|  |
| --- |
|  |
| **🔐 Best Practice: Use Service Principals for Automation Pipelines Rather Than User Tokens**  **🎯 Objective**  Ensure all automation and CI/CD pipelines use **service principals**, so you can: ✅ Avoid dependency on individual user credentials. ✅ Improve security and auditability. ✅ Enable reliable, non-expiring authentication for infrastructure and data workflows.  **🟢 Why This Matters**  Using personal user tokens in automation creates risks:   * Tokens expire or are revoked when users leave the organization. * Pipelines break unexpectedly, disrupting production workflows. * Audit logs show individual user accounts instead of clear service identities. * Compliance violations if least-privilege principles are not applied.   Service principals are **purpose-built identities** designed for non-interactive workloads.  **🟢 Recommended Approach**  ✅ **Provision a Dedicated Service Principal per Automation Use Case:** Examples:   * Infrastructure deployments (infra-deployer) * Data ingestion pipelines (data-ingest-sp) * Scheduled analytics jobs (analytics-runner)   ✅ **Assign Only the Minimum Permissions:**   * No broad account admin rights. * Limit to specific catalogs, schemas, or clusters.   ✅ **Use Secret Scopes to Store Credentials:** Never embed secrets in code or pipeline variables.  **🟢 Example Workflow**  1️⃣ **Create a Service Principal in Azure AD or Okta:**   * Name it descriptively (e.g., databricks-automation-sp). * Assign it to the Databricks application.   2️⃣ **Generate a Client Secret:**   * Store it securely in your vault (Azure Key Vault, AWS Secrets Manager).   3️⃣ **Assign Unity Catalog Permissions via Grants:**  sql  CopyEdit  GRANT USAGE ON CATALOG sales\_data TO `databricks-automation-sp`;  GRANT SELECT, INSERT ON SCHEMA sales\_data.raw TO `databricks-automation-sp`;  4️⃣ **Configure the Pipeline to Use the Service Principal:**   * Reference the secret from your CI/CD system. * Authenticate using OAuth 2.0.   ✅ **Terraform Example for Authentication:**  hcl  CopyEdit  provider "databricks" {  azure\_client\_id = var.service\_principal\_client\_id  azure\_client\_secret = var.service\_principal\_client\_secret  azure\_tenant\_id = var.tenant\_id  }  **🟢 Best Practices for Success**  ✅ **Rotate Secrets Regularly:** Automate rotation every 90 days.  ✅ **Use Least Privilege:** Grant only the permissions the pipeline requires—nothing more.  ✅ **Name Clearly:** Prefix service principals with sp- or automation- to distinguish them from human users.  ✅ **Enable Logging:** Ensure all actions performed by service principals are logged and reviewed.  ✅ **Avoid Using Personal Tokens:** Enforce policies preventing personal PATs (Personal Access Tokens) in pipelines.  **🟢 Sample Communication Announcement**  **Subject:** 🔐 Service Principals Required for All Automation Pipelines  **Body:**  As part of our security best practices, all Databricks automation and CI/CD workflows must use service principals for authentication.  **Key Benefits:**   * Credentials are independent of any individual user. * No disruptions when employees change roles or leave. * Clear audit trails and secure, revocable access.   **Important:**   * Personal access tokens (PATs) must be removed from pipeline configurations. * All service principals must follow least-privilege access policies.   For guidance, contact [Data Platform Security Team]. |

Best Practise # 7

|  |
| --- |
|  |
| **🧩 Best Practice: All Grants and Resources Declared in Code—Never Manually Created in the Console**  **🎯 Objective**  Ensure your entire Databricks environment is **repeatable, auditable, and version-controlled**, so you can: ✅ Avoid drift between workspaces. ✅ Track every change with clear ownership. ✅ Simplify disaster recovery and migrations.  **🟢 Why This Matters**  Manually creating resources (catalogs, schemas, tables) or applying grants in the UI can lead to:   * Hidden inconsistencies between regions. * Lack of visibility into who changed what. * Challenges reproducing environments in test and production. * Increased security risks from untracked permissions.   Declaring everything in code ensures **one source of truth**—your infrastructure repository.  **🟢 Recommended Approach**  ✅ **Declare All Resources in Terraform or Another IaC Tool:**   * Metastores * Catalogs * Schemas * Storage credentials * External locations * Grants and permissions   ✅ **Example Workflow:**  1️⃣ **Develop Infrastructure Code in Git:**   * Create modular Terraform configurations. * Version every change through pull requests.   2️⃣ **Validate Configurations:**   * Run terraform validate and terraform plan before applying.   3️⃣ **Deploy Through CI/CD Pipelines:**   * Use GitHub Actions, Azure DevOps, or GitLab CI. * Require approvals for production deployments.   4️⃣ **Document All Modules and Variables:**   * Maintain a central README for your infrastructure repository.   ✅ **Example Terraform Snippet:**  hcl  CopyEdit  resource "databricks\_catalog" "sales\_data" {  name = "sales\_data"  comment = "Sales line of business data catalog"  }  resource "databricks\_schema" "sales\_raw" {  name = "raw"  catalog\_name = databricks\_catalog.sales\_data.name  comment = "Raw ingestion zone"  }  resource "databricks\_grants" "sales\_raw\_grants" {  schema = databricks\_schema.sales\_raw.id  grant {  principal = "sales\_readers"  privileges = ["SELECT", "USAGE"]  }  }  **🟢 Best Practices for Success**  ✅ **Never Click “Create” in the Console:** Treat the console as **read-only** for verification.  ✅ **Review All Changes:** Require peer review for every infrastructure pull request.  ✅ **Keep Code DRY:** Use Terraform modules to avoid duplication across catalogs and schemas.  ✅ **Tag Resources:** Include metadata (e.g., environment, owner, business\_unit) to improve clarity.  ✅ **Plan Regular Audits:** Verify no manually created resources have drifted from code.  **🟢 Sample Communication Announcement**  **Subject:** 🧩 Infrastructure as Code Enforcement Policy  **Body:**  Going forward, all Databricks resources—including catalogs, schemas, and permissions—must be declared in code and deployed through our CI/CD pipelines.  **Key Benefits:**   * Full visibility and auditability. * Consistency across environments. * Simplified disaster recovery.   **Important:**   * Manual creation or modification of resources in the console is not permitted. * Please submit pull requests to the infrastructure repository for any changes.   For questions or assistance, contact [Data Platform Engineering]. |

Best Pracitse

|  |
| --- |
|  |
| **Best Practice: Centralize Audit Logs in a Dedicated Logging Workspace or SIEM**  **🎯 Objective**  Establish a **single source of truth** for all Databricks audit events, so you can: ✅ Meet compliance requirements. ✅ Quickly investigate security incidents. ✅ Maintain clear accountability across teams.  **🟢 Why This Matters**  Audit logs are your **first line of defense** for detecting:   * Unauthorized access attempts. * Data exfiltration. * Policy misconfigurations.   Without centralized logging, you risk:   * Incomplete visibility across workspaces. * Delays in forensic investigations. * Non-compliance with regulatory standards (e.g., GDPR, HIPAA).   **🟢 Recommended Approach**  ✅ **Centralize All Audit Logs in One Location:**   * A dedicated **logging workspace** in Databricks. * Or an external **Security Information and Event Management (SIEM)** system (e.g., Splunk, Azure Sentinel, AWS Security Lake).   ✅ **Example Workflow:**  1️⃣ **Enable Audit Logging in Each Workspace**   * In the Databricks admin console, configure audit log delivery to a secure cloud storage location (e.g., an S3 bucket or Azure Data Lake container).   2️⃣ **Secure the Storage:**   * Restrict access to your security and compliance teams. * Enable immutable storage if required by regulations.   3️⃣ **Ingest Logs into SIEM:**   * Use native connectors or ingestion pipelines (e.g., Azure Event Grid, AWS Lambda, Splunk forwarders). * Parse logs into structured formats for alerting and dashboards.   4️⃣ **Monitor and Alert:**   * Set up alerts for high-risk events (e.g., permission changes, failed logins, data exports). * Review dashboards regularly.   **🟢 Best Practices for Success**  ✅ **Retain Logs for At Least 1 Year:** Some compliance standards require longer retention.  ✅ **Encrypt Logs at Rest and In Transit:** Use customer-managed keys if your policies require them.  ✅ **Limit Access:** Only designated compliance and security personnel should have permissions.  ✅ **Regularly Validate Ingestion:** Test that logs are flowing end-to-end from Databricks into your SIEM.  ✅ **Document Everything:** Record:   * Where logs are stored. * How long they’re retained. * Who has access.   **🟢 Sample Communication Announcement**  **Subject:** 📝 Audit Logging Centralization Complete  **Body:**  We have configured centralized audit logging for all Databricks workspaces.  **Key Details:**   * All audit events are collected in a dedicated logging workspace and our SIEM platform. * Logs are retained for 1 year to support compliance and investigations. * Access is restricted to the Security and Compliance teams.   **Next Steps:**   * All security incidents will be monitored through the central logging system. * Please report any suspected issues to [Security Operations Center].   For questions or support, contact [Data Platform Security Team]. |

Deployments Steps in Detail

|  |
| --- |
|  |
| **🛡️ Identity & Access: Integrate AAD/Okta and Sync Groups via SCIM**  **🎯 Objectives**  By completing this phase, you will: ✅ Enable single sign-on (SSO) with your enterprise identity provider (Azure AD or Okta). ✅ Automatically provision and deprovision user and group access via SCIM. ✅ Ensure consistent, compliant, and auditable identity management across Databricks workspaces.  **🧭 Audience**   * Identity & Access Management (IAM) Administrators * Cloud Platform Engineers * Security Engineers   **🟢 1. Integrate Azure AD or Okta for Single Sign-On**  ✅ **Why integrate SSO?**   * Users log in with corporate credentials. * Centralized authentication and MFA policies. * Simplifies onboarding and offboarding.   **📘 A. Azure Active Directory (AAD)**  ✅ **Steps to integrate AAD SSO:** 1️⃣ Register an **Enterprise Application** for Databricks in Azure AD. 2️⃣ Configure SAML settings:   * **Entity ID:** Provided in the Databricks admin console. * **Reply URL (Assertion Consumer Service):** Provided in the console. 3️⃣ Upload the Azure AD SAML metadata XML to Databricks. 4️⃣ Assign users and groups to the application.   ✅ **Tip:** Use **security groups** in Azure AD to control which teams get access.  **📘 B. Okta**  ✅ **Steps to integrate Okta SSO:** 1️⃣ Create a **SAML 2.0 App Integration** in Okta. 2️⃣ Configure SAML settings with Databricks metadata:   * **Single Sign-On URL** * **Audience URI (SP Entity ID)** 3️⃣ Upload Okta metadata back into Databricks. 4️⃣ Assign Okta groups and users.   ✅ **Best Practice:** Enable **Just-In-Time (JIT) provisioning** to automatically create user records in Databricks upon first login.  **🟢 2. Enable SCIM for Group and User Sync**  ✅ **What is SCIM?** SCIM (System for Cross-domain Identity Management) automatically synchronizes users and groups between your IdP and Databricks.  ✅ **Benefits:**   * No manual user creation. * Auto-updates when group membership changes. * Automatic deprovisioning.   ✅ **Steps to enable SCIM:**  **📘 A. Azure Active Directory**  1️⃣ In your Azure AD Enterprise Application:   * Go to **Provisioning**. * Set **Provisioning Mode** to **Automatic**. 2️⃣ Enter the **SCIM endpoint URL**:   perl  CopyEdit  https://<databricks-instance>#scim/v2  3️⃣ Generate a Databricks **SCIM token** in the Admin Console:   * Navigate to **User Management > SCIM Token**. * Copy and store securely. 4️⃣ Configure the **Secret Token** in Azure AD. 5️⃣ Map attributes (default mappings are usually sufficient). 6️⃣ Test provisioning and start synchronization.   ✅ **Tip:** Set the **scope** to Sync only assigned users and groups for precise control.  **📘 B. Okta**  1️⃣ In the Okta Application:   * Enable **SCIM provisioning**. 2️⃣ Configure the SCIM connector base URL:   perl  CopyEdit  https://<databricks-instance>#scim/v2  3️⃣ Add the **SCIM token** as the authentication secret. 4️⃣ Map Okta groups to Databricks groups. 5️⃣ Enable incremental sync.  **🟢 3. Validate SCIM Sync**  ✅ **Checklist:**   * Users created in Databricks after sync. * Groups appear in **User Management > Groups**. * Group memberships match your IdP. * User removals propagate correctly (deprovisioned). * Access logs show SCIM provisioning events.   ✅ **Tip:** Test with a dedicated pilot group first.  **🟢 4. Best Practices for Success**  ✅ **Use Group-Based Access:** Assign workspace and Unity Catalog permissions **only via groups**, never individuals.  ✅ **Set Clear Naming Conventions:** Prefix groups clearly (e.g., sales\_readers, marketing\_admins).  ✅ **Enable Notifications:** Configure your IdP to alert when provisioning errors occur.  ✅ **Review Regularly:** Audit synced groups quarterly to ensure compliance.  ✅ **Secure Your SCIM Token:** Rotate SCIM tokens periodically and store them in a secure vault.  **🟢 5. Sample Communication Announcement**  **Subject:** 🛡️ Single Sign-On and Automated Group Sync Live  **Body:**  We have successfully integrated Single Sign-On and automated user/group provisioning for Databricks.  **Key Benefits:**   * Sign in using your corporate credentials (Azure AD / Okta). * Group membership automatically controls your workspace and data access. * No more manual user creation or deactivation.   **Next Steps:**   * All users will log in via SSO effective immediately. * Access changes must be managed in your corporate directory.   For assistance, contact [Identity & Access Management Team]. |

|  |
| --- |
| Storage |
| **🗄️ Storage: Create Region-Specific Storage Accounts and Define Storage Credentials**  **🎯 Objectives**  By completing this step, you will: ✅ Provision dedicated cloud storage in each region for Unity Catalog-managed and external tables. ✅ Define secure access credentials for Databricks to read and write data. ✅ Enable consistent, auditable storage across your environments.  **🧭 Audience**   * Cloud Platform Engineers * Data Platform Engineers * Security Administrators   **🟢 1. Create Region-Specific Storage Accounts or Buckets**  ✅ **Why region-specific storage?**   * Meet compliance and residency requirements. * Improve performance by keeping data close to compute. * Simplify lifecycle and cost management.   ✅ **Recommended Approach:** Create **one dedicated storage account (Azure) or bucket (AWS/GCP) per region.**   |  |  |  | | --- | --- | --- | | **Region** | **Storage Resource** | **Purpose** | | East US | datalakeeast (Azure Storage Account) | Metastore and Unity Catalog data | | West Europe | datalakeeu (Azure Storage Account) | Metastore and Unity Catalog data |   ✅ **Naming Conventions:** Use clear names like:  php-template  CopyEdit  <environment><region><purpose>  Examples:   * prduseastunifiedatalake * prdeuwestunifiedatalake   **🟢 2. Organize Containers or Folders**  ✅ **Example (Azure):** Inside each storage account, create containers:   * unitycatalog * externaldata * backups   ✅ **Example (AWS S3):** Use prefixes to organize:  arduino  CopyEdit  s3://prd-useast-unitycatalog/  s3://prd-useast-externaldata/  ✅ **Tip:** Keep managed tables separate from external datasets to avoid accidental deletions.  **🟢 3. Define Storage Credentials**  ✅ **What are Storage Credentials?** Storage credentials are secure references that Unity Catalog uses to access your storage—**not individual keys embedded in tables.**  ✅ **Azure:** Use **Service Principal** authentication:   * Register an Azure AD app. * Grant Storage Blob Data Contributor on the container. * Create a secret scope in Databricks to store the client secret securely.   ✅ **AWS:** Use an **IAM role** with:   * s3:ListBucket, s3:GetObject, s3:PutObject on the relevant bucket. * Configure a trust policy for Databricks.   ✅ **Terraform Example (Azure):**  hcl  CopyEdit  resource "databricks\_azure\_service\_principal\_secret" "storage\_creds" {  application\_id = var.app\_id  client\_secret = var.client\_secret  directory\_id = var.tenant\_id  }  resource "databricks\_storage\_credential" "unity\_storage" {  name = "unitycatalog\_us\_east"  azure\_service\_principal {  client\_id = databricks\_azure\_service\_principal\_secret.storage\_creds.application\_id  tenant\_id = databricks\_azure\_service\_principal\_secret.storage\_creds.directory\_id  client\_secret = databricks\_azure\_service\_principal\_secret.storage\_creds.client\_secret  }  }  ✅ **Terraform Example (AWS):**  hcl  CopyEdit  resource "databricks\_storage\_credential" "unity\_storage" {  name = "unitycatalog\_us\_east"  aws\_iam\_role {  role\_arn = "arn:aws:iam::123456789012:role/UnityCatalogAccess"  }  }  **🟢 4. Test Storage Credentials**  ✅ **SQL Test Command:**  sql  CopyEdit  DESCRIBE STORAGE CREDENTIAL unitycatalog\_us\_east;  ✅ **Checklist:**   * Credential created successfully. * No errors when accessing storage. * Permissions limited to necessary paths. * Logs confirm access is working.   **🟢 5. Best Practices for Success**  ✅ **Use Least Privilege:** Grant only the permissions Unity Catalog requires.  ✅ **Rotate Credentials Regularly:** Set reminders to rotate keys or secrets periodically.  ✅ **Enable Logging:** Turn on access logs in storage accounts/buckets for audit.  ✅ **Avoid Hardcoding:** Store secrets in secure vaults or secret scopes.  ✅ **Document Everything:** Record:   * Storage account names * Container paths * Credential names * Expiration dates   **🟢 6. Sample Communication Announcement**  **Subject:** 🗄️ Region-Specific Unity Catalog Storage Created  **Body:**  We have provisioned dedicated cloud storage for Unity Catalog data in each region.  **Storage Resources:**   * East US: datalakeeast Azure Storage Account * West Europe: datalakeeu Azure Storage Account   **Storage Credentials:**   * Configured using secure Service Principals (Azure) / IAM Roles (AWS)   **Next Steps:**   * External locations and managed table locations will be configured. * Access policies will be applied to enforce least-privilege access.   For any questions, contact [Data Platform Team]. |

|  |
| --- |
| Metastore Deployment and Assignment |
| **🏛️ Metastore: Deploy Metastore per Region and Assign to Workspaces**  **🎯 Objectives**  By completing this step, you will: ✅ Deploy a dedicated Unity Catalog metastore in each required cloud region. ✅ Assign each metastore to the correct set of Databricks workspaces. ✅ Prepare the environment for consistent catalog, schema, and permission management.  **🧭 Audience**   * Data Platform Engineers * Cloud Platform Engineers * Unity Catalog Administrators   **🟢 1. Plan Regional Metastore Deployment**  ✅ **Why regional separation matters:**   * Compliance (e.g., data residency regulations) * Performance optimization (data locality) * Clear operational boundaries between regions   ✅ **Recommended Approach:** Create **one metastore per region**, covering all workspaces in that geography.   |  |  |  | | --- | --- | --- | | **Region** | **Metastore Name** | **Assigned Workspaces** | | East US | metastore\_us\_east | Sales US, Marketing US, eCommerce US | | West Europe | metastore\_eu\_west | Sales EU, Marketing EU, eCommerce EU |   ✅ **Tip:** Keep metastore names **short and region-specific** to avoid confusion.  **🟢 2. Create the Metastore**  ✅ **Terraform Example:**  hcl  CopyEdit  resource "databricks\_metastore" "us\_east" {  name = "metastore\_us\_east"  storage\_root = "abfss://unitycatalog@datalakeeast.dfs.core.windows.net/"  region = "eastus"  }  ✅ **SQL CLI Example:** Alternatively, you can create the metastore manually via the Databricks CLI or REST API, but **Infrastructure as Code is recommended** for consistency.  **🟢 3. Assign the Metastore to Workspaces**  ✅ **Purpose:** This links your workspaces to the metastore so Unity Catalog is enabled.  ✅ **Terraform Example:**  hcl  CopyEdit  resource "databricks\_metastore\_assignment" "sales\_us" {  metastore\_id = databricks\_metastore.us\_east.id  workspace\_id = var.sales\_us\_workspace\_id  default\_catalog\_name = "main"  }  ✅ **Repeat for each workspace in the region:**   * Marketing workspace * eCommerce workspace   ✅ **Tip:** Always assign the **default catalog name** (usually main) to ensure smooth initial usage.  **🟢 4. Validate the Assignments**  ✅ **Checklist:**   * All workspaces show Unity Catalog enabled. * You can see the main catalog when you connect to each workspace. * Catalog and schema creation works as expected.   ✅ **Validation SQL:** Run this in each workspace:  sql  CopyEdit  SHOW CATALOGS;  ✅ **Expected Output:**  diff  CopyEdit    **🟢 5. Best Practices for Success**  ✅ **Use Consistent Naming:** Follow <prefix>\_<region> naming conventions for metastore clarity.  ✅ **Document Assignments:** Record which workspaces are mapped to which metastores in your platform wiki.  ✅ **Secure the Root Storage:** Ensure the storage\_root container has strict access controls—only Databricks and admins should have permissions.  ✅ **One Workspace = One Metastore:** A workspace can only be linked to a **single** metastore.  ✅ **Plan Ahead:** Metastore assignments can be changed later, but doing so requires **all tables and permissions to be migrated**—plan carefully.  **🟢 6. Sample Communication Announcement**  **Subject:** 🏛️ Unity Catalog Metastore Deployed & Assigned  **Body:**  We have successfully deployed Unity Catalog metastores per region and assigned them to all Databricks workspaces.  **Metastore Assignments:**   * metastore\_us\_east: Sales US, Marketing US, eCommerce US * metastore\_eu\_west: Sales EU, Marketing EU, eCommerce EU   **Next Steps:**   * Catalogs and schemas will be provisioned shortly. * Access policies will be applied in coordination with data stewards.   For questions, contact [Data Platform Engineering]. |

|  |
| --- |
| Catalogs and Schemas |
| **Catalogs and Schemas:**   * Create catalogs (sales\_data, marketing\_data, etc.). * Create schemas (raw, curated, analytics).   **🗂️ Catalogs and Schemas: Create Catalogs and Schemas**  **🎯 Objectives**  By the end of this step, you will: ✅ Establish consistent data boundaries for each line of business. ✅ Create clearly named catalogs to separate domains. ✅ Set up schemas to define data lifecycle zones. ✅ Prepare the environment for permissions and ingestion pipelines.  **🧭 Audience**   * Data Platform Engineers * Data Architects * Unity Catalog Administrators   **🟢 1. Design Your Catalog Structure**  ✅ **Recommended Catalog Naming:** Use the <business\_unit>\_data convention to keep everything clear and consistent.   |  |  | | --- | --- | | **Line of Business** | **Catalog Name** | | Sales | sales\_data | | Marketing | marketing\_data | | eCommerce | ecommerce\_data |   ✅ **Why this matters:**   * Clear ownership and responsibility. * Easier access control (scoped per catalog). * Clean separation of concerns.   **🟢 2. Design Your Schema Structure**  Inside each catalog, use **standard lifecycle schemas** to organize datasets:   |  |  | | --- | --- | | **Schema Name** | **Purpose** | | raw | Landing area for ingested data. | | curated | Cleaned and transformed datasets. | | analytics | Ready-to-consume reporting and BI datasets. |   ✅ **Tip:** Apply this pattern consistently across all catalogs to reduce confusion and training overhead.  **🟢 3. Create Catalogs**  ✅ **SQL Example:** Run these commands **once per catalog**:  sql  CopyEdit  CREATE CATALOG sales\_data  COMMENT "Sales line of business data catalog";  sql  CopyEdit  CREATE CATALOG marketing\_data  COMMENT "Marketing line of business data catalog";  sql  CopyEdit  CREATE CATALOG ecommerce\_data  COMMENT "eCommerce line of business data catalog";  ✅ **Best Practice:** Include clear comments—Unity Catalog displays them in the UI.  **🟢 4. Create Schemas**  ✅ **SQL Example:** Inside each catalog, create the standard schemas.  **Sales Example:**  sql  CopyEdit  CREATE SCHEMA sales\_data.raw  COMMENT "Raw ingestion zone for Sales";  sql  CopyEdit  CREATE SCHEMA sales\_data.curated  COMMENT "Curated clean zone for Sales";  sql  CopyEdit  CREATE SCHEMA sales\_data.analytics  COMMENT "Analytics and reporting zone for Sales";  ✅ **Repeat this structure for Marketing and eCommerce:**  sql  CopyEdit  CREATE SCHEMA marketing\_data.raw;  CREATE SCHEMA marketing\_data.curated;  CREATE SCHEMA marketing\_data.analytics;  CREATE SCHEMA ecommerce\_data.raw;  CREATE SCHEMA ecommerce\_data.curated;  CREATE SCHEMA ecommerce\_data.analytics;  ✅ **Tip:** Add comments to every schema so users understand their purpose.  **🟢 5. Verify Structure**  ✅ **Checklist:**   * All catalogs created successfully. * All schemas present in each catalog. * Comments display in Data Explorer. * No typos in names (naming consistency is critical).   ✅ **Example Validation:**  sql  CopyEdit  SHOW CATALOGS;  SHOW SCHEMAS IN CATALOG sales\_data;  SHOW SCHEMAS IN CATALOG marketing\_data;  ✅ **Expected Output Example:**  Pgsql    **🟢 6. Best Practices for Success**  ✅ **Be Consistent:** Never mix naming conventions (e.g., salesdata vs. sales\_data).  ✅ **Lock Down Empty Catalogs:** Immediately restrict access so nothing gets created ad hoc before access policies are applied.  ✅ **Document Everything:** Capture each catalog and schema’s purpose and ownership in your data platform wiki.  ✅ **Review Quarterly:** Confirm your structure still fits your data governance needs as your business evolves.  **🟢 7. Sample Communication Announcement**  **Subject:** 🗂️ Unity Catalog Structure Complete: Catalogs & Schemas  **Body:**  We have established the foundational Unity Catalog structure for Sales, Marketing, and eCommerce.  **Catalogs Created:**   * sales\_data * marketing\_data * ecommerce\_data   **Schemas in Each Catalog:**   * raw * curated * analytics   **Next Steps:**   * Access policies will be applied shortly. * All data ingestion pipelines will target the raw schemas.   For questions or assistance, contact [Data Platform Team]. |

|  |
| --- |
| Access Policy |
| **🛡️ Access Policies: Grant Privileges to Groups**  **🎯 Objectives**  When you complete this step, you will: ✅ Define which groups can access which data. ✅ Grant least-privilege access by default. ✅ Enable self-service analytics for approved users. ✅ Document and track all access grants for compliance.  **🧭 Audience**   * Data Platform Admins * Security Engineers * Data Stewards   **🟢 1. Prepare Group Structures**  Before granting any privileges, ensure you have **consistent, well-defined groups**:  ✅ **Recommended structure per line of business:**   | **Line of Business** | **Read-only Group** | **Read/Write Group** | **Admin Group** | | --- | --- | --- | --- | | Sales | sales\_readers | sales\_writers | sales\_admins | | Marketing | marketing\_readers | marketing\_writers | marketing\_admins | | eCommerce | ecommerce\_readers | ecommerce\_writers | ecommerce\_admins |   ✅ **Tip:** Provision groups via SCIM (Azure AD / Okta) so membership is managed centrally.  **🟢 2. Understand Privilege Types**  Unity Catalog privileges are **granular and explicit**.  ✅ **Common privileges you will assign:**   | **Privilege** | **What It Allows** | | --- | --- | | USAGE | See the catalog/schema exists | | SELECT | Query tables/views | | INSERT | Insert data | | UPDATE | Modify data | | DELETE | Remove data | | ALL PRIVILEGES | Full control |   ✅ **Tip:** **USAGE** must be granted at every level (catalog + schema) before other privileges work.  **🟢 3. Grant Privileges at Each Layer**  **📘 A. Catalog Level**  ✅ Example: Grant USAGE to readers  sql  CopyEdit  GRANT USAGE ON CATALOG sales\_data TO `sales\_readers`;  ✅ Grant ALL PRIVILEGES to admins  sql  CopyEdit  GRANT ALL PRIVILEGES ON CATALOG sales\_data TO `sales\_admins`;  **📘 B. Schema Level**  ✅ Grant USAGE on schema  sql  CopyEdit  GRANT USAGE ON SCHEMA sales\_data.curated TO `sales\_readers`;  ✅ Grant CREATE if writers need to create tables  sql  CopyEdit  GRANT CREATE ON SCHEMA sales\_data.curated TO `sales\_writers`;  **📘 C. Table Level**  ✅ Grant SELECT for read access  sql  CopyEdit  GRANT SELECT ON ALL TABLES IN SCHEMA sales\_data.curated TO `sales\_readers`;  ✅ Grant INSERT, UPDATE, DELETE for writers  sql  CopyEdit  GRANT INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA sales\_data.curated TO `sales\_writers`;  ✅ **Tip:** Use GRANT ON FUTURE so new tables automatically inherit privileges.  Example:  sql  CopyEdit  GRANT SELECT ON FUTURE TABLES IN SCHEMA sales\_data.curated TO `sales\_readers`;  **🟢 4. Document and Version Control Grants**  ✅ Keep a **central record** of all grants:   * Who has access * What level of privilege * When it was granted * Who approved it   ✅ **Best Practice:** Store grant definitions in **Terraform**:  **Example Terraform snippet:**  hcl  CopyEdit  resource "databricks\_grants" "sales\_curated" {  schema = "sales\_data.curated"  grant {  principal = "sales\_readers"  privileges = ["USAGE", "SELECT"]  }  grant {  principal = "sales\_writers"  privileges = ["USAGE", "SELECT", "INSERT", "UPDATE", "DELETE"]  }  }  **🟢 5. Validate Access**  ✅ **Checklist for each group:**   * Can list catalogs (SHOW CATALOGS;) * Can see schemas (SHOW SCHEMAS IN sales\_data;) * Can see tables (SHOW TABLES IN sales\_data.curated;) * Can query data (if applicable) * Cannot see data outside their scope   ✅ **Example query to test:**  sql  CopyEdit  SELECT \* FROM sales\_data.curated.orders LIMIT 10;  ✅ **Pro Tip:** Have a **test user account** in each group to confirm permissions yourself.  **🟢 6. Best Practices for Success**  ✅ **Least Privilege by Default:** Grant only what’s needed—never default to ALL PRIVILEGES for general users.  ✅ **Use Groups, Not Individuals:** This ensures scalability and avoids “snowflake” permissions.  ✅ **Automate Everything:** Manage grants through Terraform or automation pipelines.  ✅ **Document Access Requests:** Have a clear approval workflow (e.g., ServiceNow).  ✅ **Review Quarterly:** Revalidate all grants every 3 months.  **🟢 7. Sample Communication Announcement**  **Subject:** 🛡️ Unity Catalog Access Policy Updates  **Body:**  We have completed the configuration of Unity Catalog access policies for Sales, Marketing, and eCommerce.  **What this means:**   * You will use your assigned group (readers, writers, admins) to access data. * Permissions are now centrally managed—no local grants are permitted. * Requests for new access or changes must be submitted via [access request process].   For questions, contact [Data Platform Security Team].  ✅ **If you’d like help**, I can provide:   * A **Terraform module** for standardized grants * A **sample access request form** * A **step-by-step validation checklist template** |

|  |
| --- |
| Automation |
| **🤖 Automation: Write and Test Terraform Modules & Pipeline Deployment**  **🎯 Objectives**  When you complete this phase, you will have: ✅ Declarative Infrastructure as Code (IaC) for all Unity Catalog resources ✅ Reproducible, version-controlled deployments ✅ Automated pipelines to apply changes safely across environments  **🧭 Audience**   * Cloud Platform Engineers * Data Platform Engineers * DevOps Teams * Data Architects   **🟢 1. Write Terraform Modules**  **📘 A. Plan Your Module Structure**  ✅ **Recommended modular approach:**   * metastore module:   + Metastore resource   + Assignments to workspaces * storage module:   + Storage credentials   + External locations * catalogs module:   + Catalog and schema definitions * access module:   + Grants for groups and service principals   ✅ **Folder structure example:**  pgsql  CopyEdit  terraform/  modules/  metastore/  storage/  catalogs/  access/  environments/  dev/  prod/  **📘 B. Define Naming Conventions**  ✅ Use consistent names:   * Metastore: sales\_us\_east * Catalog: sales\_data * Schemas: raw, curated, analytics * Locations: abfss://datalake@yourstorageaccount/...   **📘 C. Example: Metastore Module**  **main.tf:**  hcl  CopyEdit  resource "databricks\_metastore" "this" {  name = var.metastore\_name  storage\_root = var.storage\_root  region = var.region  }  **variables.tf:**  h  CopyEdit  variable "metastore\_name" {}  variable "storage\_root" {}  variable "region" {}  ✅ **Tip:** Use variables for everything—avoid hardcoding paths or names.  **📘 D. Example: Catalog Module**  **main.tf:**  hcl  CopyEdit  resource "databricks\_catalog" "this" {  name = var.catalog\_name  comment = var.comment  properties = var.properties  }  ✅ **Tip:** Include metadata tags for lineage.  **🟢 2. Test Terraform Modules**  ✅ **Steps:**  1️⃣ **Validate Syntax**  bash  CopyEdit  terraform init  terraform validate  2️⃣ **Plan Deployment**  bash  CopyEdit  terraform plan -var-file="environments/dev/terraform.tfvars"  3️⃣ **Apply to Dev**  bash  CopyEdit  terraform apply -auto-approve -var-file="environments/dev/terraform.tfvars"  4️⃣ **Verify Resources**   * Check in Databricks Admin Console * Confirm catalogs and schemas are created * Run SHOW GRANTS to confirm permissions   ✅ **Best Practice:** Use **Terraform workspaces** or **separate state files** for dev and prod.  **🟢 3. Pipeline Deployment**  **📘 A. Choose Your Pipeline Tool**  ✅ Options:   * Azure DevOps Pipelines * GitHub Actions * GitLab CI/CD * Jenkins   ✅ **Recommendation:** Use GitHub Actions for simple workflows or Azure DevOps if you need integration with Azure RBAC.  **📘 B. Example GitHub Actions Workflow**  **.github/workflows/deploy.yml:**  yaml  CopyEdit  name: Deploy Unity Catalog  on:  push:  branches:  - main  jobs:  deploy:  runs-on: ubuntu-latest  steps:  - name: Checkout  uses: actions/checkout@v3  - name: Setup Terraform  uses: hashicorp/setup-terraform@v2  - name: Terraform Init  run: terraform init  - name: Terraform Validate  run: terraform validate  - name: Terraform Plan  run: terraform plan -out=tfplan  - name: Terraform Apply  run: terraform apply -auto-approve tfplan  ✅ **Tip:** Use environment secrets for Databricks tokens and cloud credentials.  **📘 C. Approvals and Controls**  ✅ **Implement:**   * Pull request reviews before merging to main * Manual approval gates before production deploy * State file storage in secure backend (e.g., Azure Storage Account, S3)   **📘 D. Post-Deployment Validation**  ✅ Checklist after every pipeline run:   * Confirm all resources in Databricks match the Terraform state * Run test queries to confirm access * Verify audit logs show expected events   **🟢 4. Best Practices for Success**  ✅ **Write modules once—reuse everywhere.** ✅ **Never apply manually—use pipelines.** ✅ **Keep state files secure and versioned.** ✅ **Separate dev, staging, and prod environments.** ✅ **Document module inputs and outputs clearly.**  **🟢 5. Sample Communication Announcement**  **Subject:** 🤖 Databricks Unity Catalog Automation Pipeline Live  **Body:**  We’ve implemented Terraform-based automation for Unity Catalog resources.  From now on:   * All changes must go through pull requests and the pipeline. * No manual console changes are permitted.   **Key Benefits:**   * Consistent, auditable deployments * Secure access controls * Faster provisioning   For any questions, contact [Data Platform Engineering]. |

|  |
| --- |
| Training Plan |
| **Training Plan: Educate Users on Naming Conventions and Access**  **🎯 Objectives**  When you complete this training, users should be able to: ✅ Understand and use the correct catalog and schema names. ✅ Know how to locate, query, and write data properly. ✅ Understand access permissions and request processes. ✅ Avoid common mistakes like writing data into raw zones or unapproved locations.  **🧭 Audience**   * Data Analysts * Data Engineers * BI Developers * Data Scientists * Business Users * Data Stewards   **🟢 1. Training Content Areas**  Below are the **key topics you must cover**:  **📘 A. Introduction to Unity Catalog**   * What is Unity Catalog? (Short explanation) * Why are we using it? (Governance, security, audit) * How is it different from legacy Hive Metastore?   **📘 B. Naming Conventions**  ✅ **Explain the standard structure:**  php-template  CopyEdit  <catalog>.<schema>.<table>  **Examples:**   * sales\_data.curated.orders * marketing\_data.analytics.campaign\_metrics * ecommerce\_data.raw. clickstream\_events   ✅ **Best Practices to Teach:**   * Always use fully qualified names (catalog.schema.table) * Never create ad-hoc tables in the default catalog * Use singular table names where possible (order, not orders if you prefer consistent naming) * Avoid spaces or special characters in names * Keep names lowercase   ✅ **Quick Reference Table (provide as a handout):**   | **Environment** | **Catalog** | **Schemas** | | --- | --- | --- | | Sales | sales\_data | raw, curated, analytics | | Marketing | marketing\_data | raw, curated, analytics | | eCommerce | ecommerce\_data | raw, curated, analytics |   **📘 C. Access and Permissions**  ✅ **Explain Roles & Groups:**   * You are assigned to one or more groups:   + sales\_readers   + marketing\_writers   + ecommerce\_admins * Access is managed centrally—no local grants   ✅ **Teach these principles:**   * **USAGE privilege** is needed on catalogs and schemas before you can see any tables * **SELECT privilege** is needed to query * **INSERT/UPDATE privileges** are needed to write data * All requests for additional access go through a formal request process (e.g., ServiceNow ticket)   ✅ **Show users how to check their access:**  **Example SQL:**  sql  CopyEdit  SHOW GRANTS ON SCHEMA sales\_data.curated;  ✅ **Demonstrate what access errors look like:**   * Permission denied error messages * What to do when you see them   **📘 D. Data Zones**  ✅ **Define zones clearly:**   * **Raw schema:** No direct writes except ingestion processes * **Curated schema:** Cleansed datasets—authorized writes * **Analytics schema:** BI-ready tables—mostly read-only   ✅ **Policy:**   * Never query or join directly on raw data without a processing step * No ad-hoc changes in curated or analytics schemas   **📘 E. Query Examples and Tools**  ✅ Show:   * How to browse catalogs and tables in Data Explorer * How to run queries in SQL Editor * How to connect BI tools (Power BI, Tableau)   ✅ Provide example queries:  sql  CopyEdit  SELECT \* FROM sales\_data.analytics.orders LIMIT 10;  ✅ Show how to save queries or create views  **📘 F. Data Sharing and Collaboration**  ✅ Explain:   * How to share data within your business unit * Why cross-catalog access is restricted * Who to contact for cross-domain access   **🟢 2. Delivery Methods**  **Recommended formats:** ✅ **Live Webinars / Workshops**   * 60–90 minutes each line of business * Interactive demo   ✅ **Recorded Video Tutorials**   * Short modules (5–10 minutes each)   ✅ **Written Quick Reference Guides**   * PDF handouts * Cheat sheets   ✅ **Internal Wiki Pages**   * FAQs * Step-by-step screenshots   ✅ **Office Hours**   * Weekly drop-in sessions for questions   **🟢 3. Example Agenda for a 90-Minute Training Session**   | **Time** | **Topic** | | --- | --- | | 0–10 min | Intro to Unity Catalog | | 10–25 min | Naming Conventions | | 25–45 min | Access Permissions & Groups | | 45–60 min | Data Zones & Policies | | 60–75 min | Live Demo: Browsing & Querying | | 75–85 min | Common Errors & How to Resolve | | 85–90 min | Q&A |   **🟢 4. Best Practices for Success**  ✅ **Keep it simple.** Avoid overwhelming users with advanced features all at once.  ✅ **Use real examples.** Show data that is familiar to them.  ✅ **Provide clear escalation paths.** Who to contact if access is wrong.  ✅ **Record everything.** Make videos and guides available 24/7.  ✅ **Follow up.** Run refresher sessions every quarter.  ✅ **TIP:** **Always give users a sandbox schema** for experimentation so they don’t risk production data.  **🟢 5. Sample Communication Announcement**  **Subject:** 🎓 Training: New Databricks Unity Catalog—What You Need to Know  **Body:**  We are rolling out Unity Catalog for secure, governed access to data across Sales, Marketing, and eCommerce.  All users are required to attend training to understand:   * Where your data lives * How to access it safely * Naming conventions and permissions   **Session Options:**   * July 10: Sales Training * July 11: Marketing Training * July 12: eCommerce Training   Please register via the link below.  For any questions, contact [Data Platform Team]. |

|  |
| --- |
| **Go-Live:**   * Enable Unity Catalog in production workspaces. * Monitor initial usage. |
| **Enable Unity Catalog in Production Workspaces**  Below is a **step-by-step process**, assuming you already have:   * A Databricks Account Console login * Your metastores created * Workspace(s) provisioned   **🎯 Step 1: Assign Metastore to Workspace**  **What this does:** Binds your workspace to a Unity Catalog metastore so all tables, schemas, and permissions are governed centrally.  **How to do it:**   1. **Log into the Databricks Account Console** (accounts.cloud.databricks.com) 2. In the left navigation, click **Workspaces** 3. Find your production workspace, click the workspace name. 4. In the **Metastore assignment** section, click **Assign metastore** 5. Select the appropriate metastore (e.g., sales\_us\_east) 6. Click **Assign**   ✅ **Tip:** If your workspace is already assigned to a metastore, review and confirm the correct one.  **🎯 Step 2: Configure Storage Credentials and External Locations**  Unity Catalog requires **external locations** for storing managed tables and external tables.  **How to do it:**   1. In Databricks, go to **Data** > **External Locations** 2. Click **Create External Location**    * **Name:** sales\_data\_location    * **URL:** e.g., abfss://databricks-sales@yourstorageaccount.dfs.core.windows.net    * **Storage Credential:** Choose or create one linked to your Azure AD identity or service principal 3. Test connection 4. Save   ✅ **Tip:** **Set default storage location** for the catalog to avoid confusion when creating tables.  **Databricks itself does not provide its own native storage layer.**  Instead, **Databricks always leverages your cloud object storage** (Azure Data Lake Storage, AWS S3, or GCP GCS).  When you hear about *managed tables* and *external tables* in Unity Catalog, here’s what’s happening behind the scenes:  **🎯 Step 3: Grant Access**  Before users can see or query data:   1. Log into a Databricks SQL warehouse (or notebook) as a **Metastore Admin** 2. Create grants for your groups:   sql  CopyEdit  GRANT USAGE ON CATALOG sales\_data TO `sales\_readers`;  GRANT USAGE ON SCHEMA sales\_data.analytics TO `sales\_readers`;  GRANT SELECT ON ALL TABLES IN SCHEMA sales\_data.analytics TO `sales\_readers`;   1. Test with representative users (use impersonation or login as a test account).   **🎯 Step 4: Validate Workspace Settings**   1. In the workspace **Admin Settings**, verify:    * Unity Catalog is enabled.    * Table access control is enforced.    * Credential passthrough is configured if needed. 2. Configure cluster policies so only Unity Catalog-compliant clusters can be launched:    * **Enable Unity Catalog** (checkbox)    * **Enforce single-user access mode** for production workloads.   ✅ **Best Practice:** Block “legacy” cluster types to avoid confusion.  **🎯 Step 5: Communicate to Users**  Before you “flip the switch,” notify:   * Data Engineers * Analysts * Stewards   Provide:   * New catalog/schema naming conventions * Example queries * Documentation links   **✅ Monitor Initial Usage**  Once Unity Catalog is live, you must **observe adoption, detect misconfigurations, and validate compliance.**  **🎯 Step 1: Enable Audit Logging**   1. Go to the **Account Console** 2. Navigate to **Audit Logs** 3. Ensure logs are exported to:    * Azure Event Hub    * AWS S3    * GCP Pub/Sub 4. Validate events appear (e.g., grant, select, create\_table)   ✅ **Tip:** Set retention policies as required by compliance (e.g., 1 year).  **🎯 Step 2: Monitor Permissions Usage**  **How to do it:**   * In a SQL Warehouse, run queries:   sql  CopyEdit  SHOW GRANTS ON SCHEMA sales\_data.analytics;   * Review who has access and adjust grants as needed.   **🎯 Step 3: Validate Data Creation & Access**  **Checklist:**   * Can users create tables? * Are tables created in the correct catalog/schema? * Are tables stored in the correct external location? * Are readers able to query data? * Are unauthorized users blocked?   **🎯 Step 4: Monitor Query Volumes and Cluster Usage**  **In the Workspace UI:**   * **Compute** > Monitor clusters to ensure Unity Catalog-compliant clusters are used. * **SQL** > Query History to check workloads.   ✅ **Best Practice:** Flag any high-volume extracts or unusual query patterns early.  **🎯 Step 5: Collect User Feedback**  **Schedule reviews:**   * After 1 week * After 1 month   Gather feedback on:   * Naming conventions * Access experience * Performance   Adjust grants, policies, and training materials accordingly.  **🎯 Step 6: Formal Sign-Off**  Once all validations pass:   * Document the acceptance criteria. * Record approvals from Data Owners and Security. * Transition the project to steady-state support.   **✅ Summary of Both Steps**  **Enable Unity Catalog in production workspaces** means:   * Assigning the metastore * Configuring storage and access * Enforcing cluster policies * Communicating changes   **Monitor initial usage** means:   * Ensuring logging is on * Checking permissions * Validating workloads * Collecting and acting on feedback |

FAQS

|  |
| --- |
|  |
| o datarbricks do not have its own storage right "external locations for storing managed tables and external tables"  **ChatGPT said:**  Excellent question—let’s clarify this carefully, because this is a very common point of confusion when people first work with Databricks and Unity Catalog:  ✅ **Correct:** **Databricks itself does not provide its own native storage layer.**  Instead, **Databricks always leverages your cloud object storage** (Azure Data Lake Storage, AWS S3, or GCP GCS).  When you hear about *managed tables* and *external tables* in Unity Catalog, here’s what’s happening behind the scenes:  **📘 1️⃣ What are Managed Tables?**   * **Definition:** Tables where Databricks manages the **lifecycle of files** (the table’s data and metadata). * **Storage location:** The files still live in **your cloud storage account**, *not* inside Databricks itself. * **Example:** If you create a managed table in a catalog called sales\_data, Databricks will store the Parquet/Delta files in a designated container or bucket you provided (via storage credentials + external locations). * **Lifecycle:** If you DROP TABLE, the files are deleted automatically.   **📘 2️⃣ What are External Tables?**   * **Definition:** Tables where you register the location of the data explicitly (e.g., an existing folder of Parquet files). * **Storage location:** Also in your cloud storage (ADLS/S3/GCS). * **Lifecycle:** If you DROP TABLE, the data files themselves **are not deleted**.   **🎯 Why Use External Locations and Storage Credentials?**  **Because Databricks must have explicit permissions** to read/write in your storage accounts. This design gives you:   * Full control over where data sits. * The ability to enforce security boundaries. * Clear separation of compute (Databricks clusters) and storage (your cloud accounts).   **⚠️ So What Does Databricks Provide?**  **Databricks provides:**   * The compute platform (clusters, SQL warehouses). * The metadata layer (Unity Catalog Metastore). * The Delta Lake transaction protocol. * The query engine and notebooks.   **But it does NOT provide:**   * Native, proprietary object storage. * Automatic hosting of your data without configuring storage credentials.   ✅ **Example in Azure:** If you create a Unity Catalog managed table:   * The files land in your Azure Data Lake Storage Gen2 account (e.g., abfss://datalake@mycompanystorage.dfs.core.windows.net/unitycatalog/sales\_data/analytics/...) * Databricks tracks the table metadata in Unity Catalog. * You control the underlying container and can apply your own storage policies.   **In short:**  **Databricks is a compute and governance platform that always relies on your own cloud object storage for actual data storage.** |
| but snowflake having its own storage right  **ChatGPT said:**  ✅ **Yes—exactly. You’ve spotted one of the *biggest differences* between Databricks and Snowflake.**  Let’s break this down clearly:  **🟢 Snowflake: Storage + Compute + Metadata = Fully Integrated**  When you load data into Snowflake:   * The files go into **Snowflake-managed storage**, which Snowflake fully controls behind the scenes. * You don’t see an Azure blob container or an S3 bucket. * Snowflake automatically manages:   + File organization   + Compression   + Encryption   + Lifecycle (e.g., time travel, failsafe) * The storage cost is charged by Snowflake as **Snowflake storage consumption**. * You can also create **external tables**, referencing your own S3/ADLS/GCS buckets, but this is optional.   ✅ **Key point:** **If you simply do COPY INTO my\_table FROM @my\_stage, your data physically moves into Snowflake’s own internal storage.**  **🟢 Databricks: Compute + Metadata Only (Storage is Yours)**  When you use Databricks:   * You must provide your own storage account/bucket. * Databricks only organizes the metadata (Unity Catalog) and runs compute. * You pay your cloud provider (Azure, AWS, GCP) directly for storage usage. * Nothing is hidden—your files stay in your bucket/container.   ✅ **Key point:** **Databricks never takes ownership of your data storage. You always retain control and pay for it separately.**  **🎯 Why This Matters**   | **Feature** | **Snowflake** | **Databricks** | | --- | --- | --- | | Where data is stored | Snowflake-managed internal storage | Your cloud object storage account (S3/ADLS/GCS) | | Who controls storage | Snowflake | You | | Who bills for storage | Snowflake (as part of Snowflake invoice) | Azure/AWS/GCP | | Can you see the raw files? | No (Snowflake abstracts this) | Yes (you can browse your buckets) | | External tables support? | Yes (optional) | Yes (must configure locations & credentials) |   **Quick analogy:**   * **Snowflake** is like checking into a hotel:   You hand over your luggage, and they store it in their secured facility. You don’t see the storage room.   * **Databricks** is like renting a storage locker:   You rent the unit (compute), but you bring your own shelves (storage account) and lock.  ✅ **That’s why when designing Databricks Unity Catalog, you have to configure:**   * **Storage Credentials** * **External Locations** * **Access Policies on your buckets**   This step does not exist in Snowflake (for internal storage) because Snowflake fully abstracts it. |
| register the location of the data explicitly means external tables , register the location databricks unity catalog are managed tables. indirectly databricks leverage the cloud platform  **ChatGPT said:**  ✅ You’re on the right track—let’s clear up the exact wording so you don’t get tripped up by the terminology:  **🟢 Unity Catalog Tables Recap**  In **Databricks Unity Catalog**, you have **two types of tables**:  **1️⃣ Managed Tables**  ✅ **Definition:** Databricks **manages the lifecycle of the data files**.   * When you create the table, Databricks writes the files into a storage location that you configured as the *default location* for the catalog (using external locations). * When you DROP TABLE, Databricks deletes the files automatically.   **You don’t have to specify the path each time you create a managed table—Databricks handles that.**  **Example:**  sql  CopyEdit  CREATE TABLE sales\_data.analytics.orders (  order\_id STRING,  amount DOUBLE  )  USING DELTA;  Here, the data lands in:  php-template  CopyEdit  abfss://<container>@<account>.dfs.core.windows.net/unity-catalog/<catalog>/<schema>/<table>/  but *you don’t need to declare the path in the CREATE TABLE statement.*  ✅ **BUT:** Even though Databricks manages the lifecycle, **the files still physically live in your cloud storage account (ADLS, S3, or GCS)**.  This is what you meant by *“Databricks leverages the cloud platform”*—exactly correct.  **2️⃣ External Tables**  ✅ **Definition:** You **explicitly register the location of existing data** in your storage account.   * You must specify the path to the data when creating the table. * Dropping the table **does NOT** delete the files.   **Example:**  sql  CopyEdit  CREATE TABLE sales\_data.analytics.orders\_external  USING DELTA  LOCATION 'abfss://mycontainer@myaccount.dfs.core.windows.net/external-data/orders/';  **You are telling Unity Catalog:**  “Here is where my table lives. Just track the metadata.”  **🔑 Key Differences Recap**   | **Feature** | **Managed Table** | **External Table** | | --- | --- | --- | | Who manages file lifecycle? | Databricks | You | | Must you specify the path? | No (default catalog location is used) | Yes | | Are files in your cloud account? | Yes | Yes | | Drop table deletes data? | Yes | No |   **✅ So Your Statement Summarized**  **“Databricks leverages the cloud platform.”** Exactly—**Databricks never owns or hosts your storage infrastructure.** It only organizes how your data is laid out and accessed.  **In simpler terms:**   * **Managed tables:** Databricks organizes your data in your cloud storage and cleans it up automatically. * **External tables:** You tell Databricks where the data is, and you manage the files yourself.   If you want, I can show an example of:   * How to create both types of tables step by step. * How they appear in Unity Catalog. * How permissions differ.   Just say the word!  **You said:**  so we canot delte external table like managed table in databricks  **ChatGPT said:**  Great question—let’s be precise here so there’s no confusion.  ✅ **You *can* delete external tables in Databricks Unity Catalog.** But what happens **when you delete them** is different compared to managed tables:  **🟢 When You Drop a Managed Table**   * **Command:**   sql  CopyEdit  DROP TABLE sales\_data.analytics.my\_managed\_table;   * **Effect:**   + Databricks deletes the **metadata** (the table definition).   + Databricks also deletes the **underlying data files** from your storage account. * **Outcome:**   Data and metadata both gone.  **🟢 When You Drop an External Table**   * **Command:**   sql  CopyEdit  DROP TABLE sales\_data.analytics.my\_external\_table;   * **Effect:**   + Databricks deletes the **metadata only** (Unity Catalog no longer tracks the table).   + The **data files remain intact** in the storage location you specified when you created the table. * **Outcome:**   Metadata gone, but files still exist in your storage bucket/container.  ✅ **So in plain language:**  You can delete the table definition (the catalog entry) just fine, but dropping an external table **does not delete the actual data files**.  **📝 Why?**  This is by design:   * You, not Databricks, own that storage path. * Databricks does not assume it should clean up files it didn’t create. * This prevents accidental loss of important datasets.   ✅ **Example Scenario:**  Imagine you have external data in:  bash  CopyEdit  abfss://shared@company.dfs.core.windows.net/finance\_data/  and you create an external table pointing there:  sql  CopyEdit  CREATE TABLE finance.external\_sales  USING DELTA  LOCATION 'abfss://shared@company.dfs.core.windows.net/finance\_data/';  **If you later run:**  sql  CopyEdit  DROP TABLE finance.external\_sales;  **Result:**   * The table disappears from Unity Catalog. * But the Parquet or Delta files in finance\_data/ are still sitting untouched in your Data Lake.   ✅ **Contrast:**  If it had been a managed table:   * Same DROP TABLE would remove the data files too.   **🎯 Quick Rule of Thumb**   | **Action** | **Managed Table** | **External Table** | | --- | --- | --- | | DROP TABLE | Deletes metadata and files | Deletes metadata only | | Data files remain? | No | Yes |   **Bottom line:**  You *can* delete an external table—but that deletion only affects the metadata in Unity Catalog, not the underlying files. |