**Databricks Platform Components—Logical Flow Centred on Unity Catalog**

**Delta Lake**

*What it is:*  
The storage layer enabling ACID transactions, time travel, and schema enforcement on data stored in your cloud object storage.

*Why it’s first:*  
All data tables are **Delta tables**—Unity Catalog uses them as the foundation.

**Unity Catalog**

*What it is:*  
The unified governance layer for:

* Centralized metadata
* Fine-grained access control
* Lineage
* Region-specific metastores

*Why it’s second:*  
You can’t secure or organize tables without Unity Catalog (or Hive metastore).

**Metastore**

*What it is:*  
Stores metadata about catalogs, schemas, tables, and views.

* Unity Catalog: **Account-level metastore** supporting multi-workspace sharing.
* Hive metastore: legacy, workspace-scoped.

**Identity and Access Management**

🔹 *What it is:*  
Defines **who can do what**:

* User and group provisioning via SCIM (AAD, Okta)
* Service principals for automation
* Grants in Unity Catalog for secure data access
* Credential passthrough to cloud storage

*Why here:*  
Once data is organized and cataloged, you must control access.

**External Data Sources and Storage Integrations**

*What it is:*  
Connections to:

* S3, ADLS, GCS (external locations)
* Partner data platforms (Snowflake, BigQuery)
* Credential configurations

*Why here:*  
External locations are **registered in Unity Catalog**, and governed through it.

**Clusters**

*What it is:*  
Compute resources to run jobs, notebooks, SQL queries:

* Interactive clusters
* Job clusters
* SQL warehouses

*Why here:*  
Compute must respect Unity Catalog permissions and credential passthrough.

**Databricks SQL**

*What it is:*  
Query engine for analytics:

* SQL warehouses
* Dashboards
* BI integrations

*Why here:*  
SQL workloads are a primary consumer of Unity Catalog-managed data.

**Workspace**

*What it is:*  
The collaborative environment containing:

* Notebooks
* Repos
* Jobs
* MLflow experiments

*Why here:*  
Developers build workflows that **consume data governed by Unity Catalog.**

**Jobs**

*What it is:*  
Orchestration layer to schedule:

* ETL pipelines
* Notebook runs
* Batch jobs

*Why here:*  
Jobs operationalize the consumption of Unity Catalog data.

**Databricks Machine Learning**

*What it is:*  
Integrated ML tools:

* Feature Store
* MLflow tracking
* AutoML

*Why here:*  
ML pipelines use Unity Catalog as the **source of governed feature data**.

**Logical Summary of the Flow:**

**Data Storage:** Delta Lake  
**Data Governance:** Unity Catalog  
**Metadata Layer:** Metastore  
**Identity & Access:** IAM and SCIM  
**External Integration:** Registered locations  
**Compute:** Clusters  
 **SQL Analytics:** Databricks SQL  
**Development:** Workspace  
**Orchestration:** Jobs  
**Machine Learning:** Databricks ML

**Databricks Logical Architecture Centered on Unity Catalog**

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Component** | **Role & Purpose** | **How It Connects to Unity Catalog** |
| **1** | **Delta Lake** | ACID storage layer for all tables. | Tables registered and governed in Unity Catalog. |
| **2** | **Unity Catalog** | Central governance layer: catalogs, schemas, permissions, lineage. | Core of all data access and metadata management. |
| **3** | **Metastore** | Stores metadata definitions (catalogs, schemas, tables, views). | Unity Catalog provides a region-scoped account-level metastore. |
| **4** | **Identity & Access Management** | Controls user, group, and service principal permissions. | Manages grants and access controls declared in Unity Catalog. |
| **5** | **External Data Sources & Locations** | Registered cloud storage and partner data connections. | External locations secured and referenced in Unity Catalog. |
| **6** | **Clusters** | Compute engines to run workloads (interactive, jobs, SQL warehouses). | All compute enforces Unity Catalog permissions and credential passthrough. |
| **7** | **Databricks SQL** | Query engine for BI, dashboards, and ad hoc analytics. | SQL Warehouses read/write governed data assets in Unity Catalog. |
| **8** | **Workspace** | Developer environment with notebooks, repos, dashboards, MLflow. | Users build and share workflows on Unity Catalog-governed data. |
| **9** | **Jobs** | Scheduled and orchestrated workloads for ETL, reporting, and processing. | Jobs reference tables, schemas, and credentials defined in Unity Catalog. |
| **10** | **Databricks Machine Learning** | Tools for feature engineering, AutoML, model training, and tracking. | Feature Store and MLflow integrated with Unity Catalog-managed datasets. |

**How to Use This Table:**

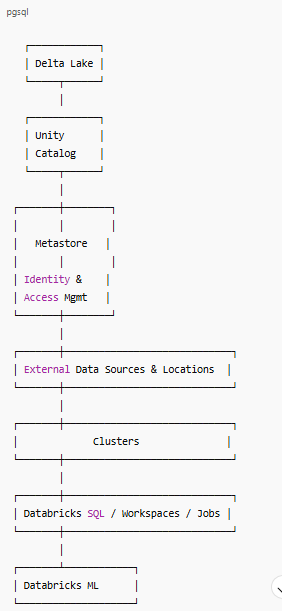
* **Columns 1–2:** Show the logical progression of components.
* **Column 3:** Explains each component’s purpose.
* **Column 4:** Shows exactly how it **relates back to Unity Catalog**.

**Databricks Logical Architecture Centered on Unity Catalog**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Component** | **Role & Purpose** | **How It Connects to Unity Catalog** | **Example Code** |
| **1** | **Delta Lake** | **ACID storage layer for all tables.** | **Tables are registered and governed as Unity Catalog managed or external tables.** | **Python**  **df.write.format("delta") .save("/mnt/raw/sales\_data")** |
| **2** | **Unity Catalog** | **Central governance layer providing catalogs, schemas, fine-grained access control, and lineage.** | **Core of all data access, security policies, and metadata management.** | **Sql**  **CREATE CATALOG sales\_data;** |
| **3** | **Metastore** | **Stores metadata definitions (catalogs, schemas, tables, views).** | **Unity Catalog provides a region-scoped account-level metastore.** | ***(Managed in admin console or via Terraform)*** |
| **4** | **Identity & Access Management** | **Manages users, groups, and service principals; controls entitlements and workspace access.** | **Grants, permissions, and credential passthrough policies are enforced centrally.** | **Sql**  **GRANT USAGE ON CATALOG sales\_data TO `sales\_team`;** |
| **5** | **External Data Sources & Locations** | **Integrates cloud storage (S3, ADLS) and partner platforms (Snowflake, BigQuery).** | **Registered external locations are secured and referenced in Unity Catalog.** | **Sql**  **CREATE EXTERNAL LOCATION sales\_ext URL 's3://bucket/data/' WITH (STORAGE CREDENTIAL my\_cred);<br>** |
| **6** | **Clusters** | **Compute engines to run workloads, including interactive clusters, job clusters, and SQL warehouses.** | **All compute respects Unity Catalog permissions and credential passthrough settings.** | ***(Created via UI or Terraform)*** |
| **7** | **Databricks SQL** | **SQL query engine for BI, dashboards, and ad hoc analytics.** | **SQL Warehouses access governed data assets defined in Unity Catalog.** | **Sql**  **SELECT \* FROM sales\_data.raw.transactions;** |
| **8** | **Workspace** | **Developer environment with notebooks, Repos, dashboards, and MLflow experiments.** | **Users develop workflows consuming data governed by Unity Catalog.** | ***(Python/SQL notebooks referencing governed tables)*** |
| **9** | **Jobs** | **Scheduling and orchestration for ETL pipelines, batch workloads, and reporting tasks.** | **Jobs reference Unity Catalog tables, schemas, and credentials for secure access.** | **Python**  **dbutils.notebook.run("/ETL/LoadSalesData", 600)** |
| **10** | **Databricks Machine Learning** | **Tools for feature engineering, AutoML, model training, and MLflow tracking.** | **Feature Store and MLflow integrated with Unity Catalog-managed datasets.** | **Python**  **import mlflow**  **mlflow.start\_run()**  **# log metrics** |

**How to use:**

* These examples are **illustrative snippets** to show what each step looks like.
* You can ada



**Databricks End-to-End Workflow with Code Samples**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Component** | **Role in Workflow** | **Sample Code / Command** |  |
| **1** | **Delta Lake** | **Store ingested raw data in Delta format with ACID transactions.** | **Python**  **df = spark.read.csv("abfss://raw@storageacct.dfs.core.windows.net/sales/\*.csv", header=True)**  **df.write.format("delta")**  **.save("/mnt/sales/raw/")** | **This just physically writes Delta files:**   * Parquet + transaction logs (\_delta\_log) to your mounted storage location. * No table has been registered yet. * No Unity Catalog metadata is created automatically. * No permissions are assigned.   **You have Delta Lake files, but Unity Catalog doesn’t know anything about them yet.** |
| **2** | **Unity Catalog** | **Create governed catalogs, schemas, and register Delta tables.** | **Sql**  **CREATE CATALOG sales\_data;**  **CREATE SCHEMA sales\_data.raw;**  **USE SCHEMA sales\_data.raw;**  **CREATE TABLE transactions**  **USING DELTA LOCATION '/mnt/sales/raw/';** | Why?  After you have saved the Delta data, you must **register it as a table** so Unity Catalog can:   * Track metadata (columns, schema, location). * Apply permissions. * Enable lineage. * Allow BI tools or Databricks SQL to query it. |
| **3** | **Metastore** | **Maintain metadata about all tables and views centrally.** | ***(Created and assigned via Databricks Admin Console or Terraform)*** |  |
| **4** | **Identity & Access Management** | **Define fine-grained access controls for teams (e.g., Sales Analysts).** | **Sql**  **GRANT SELECT ON TABLE sales\_data.raw.transactions TO `sales\_analyst\_group`;** |  |
| **5** | **External Data Sources & Locations** | **Register the cloud storage container as an external location for governed access.** | **Sql**  **CREATE STORAGE CREDENTIAL my\_credential WITH AZURE\_MANAGED\_IDENTITY 'my-managed-identity';**  **CREATE EXTERNAL LOCATION raw\_sales\_loc URL 'abfss://raw@storageacct.dfs.core.windows.net/sales/' WITH (STORAGE CREDENTIAL my\_credential);** |  |
| **6** | **Clusters** | **Run compute workloads for ingestion, transformation, and curation.** | ***(Created via Databricks UI, Terraform, or API—configured with Unity Catalog support)*** |  |
| **7** | **Databricks SQL** | **Query curated data for reporting and dashboards.** | **Sql**  **SELECT customer\_id, SUM(amount) AS total\_sales**  **FROM**  **sales\_data.raw.transactions**  **GROUP BY customer\_id;** |  |
| **8** | **Workspace** | **Develop notebooks for ETL pipelines, curation, and validation.** | **Python**  **# Example notebook code to transform**  **datadf\_clean = df.filter("amount > 0")**  **df\_clean.write.format("delta").mode("overwrite")**  **.save("/mnt/sales/curated/")** |  |
| **9** | **Jobs** | **Schedule ingestion and transformation pipelines automatically.** | **Python**  **dbutils.notebook.run("/ETL/CurateSalesData", 600)** |  |
| **10** | **Databricks Machine Learning** | **Use curated data for feature engineering and ML modeling.** | **Python**  **from databricks.feature\_store**  **import FeatureStoreClientfs = FeatureStoreClient()fs.create\_table(name="sales\_features", primary\_keys=["customer\_id"],**  **schema=df\_clean**  **.schema()**  **, description="Features for sales prediction")** |  |

Steps for Creating Metastore

|  |
| --- |
|  |
| **Steps to Set Up a Unity Catalog Metastore**  **Create the Metastore**   * Go to the **Databricks Admin Console** under **Data** or **Unity Catalog**. * Click **Create Metastore**. * Provide:   + **Name** (e.g., primary-metastore)   + **Region** (must match your cloud storage region)   + **Storage Root Location** (e.g., an S3 bucket, ADLS container, or GCS bucket where Unity Catalog stores managed table data)   **Tip:** This storage root must be **dedicated only to Unity Catalog**.  **Configure Storage Credentials**   * Create **storage credentials** to allow Databricks to access the cloud storage location.   + For AWS: an IAM role.   + For Azure: Managed Identity or Service Principal.   + For GCP: Service Account. * Assign the credentials to the Metastore.   **Assign the Metastore to Workspaces**   * In the Admin Console, **assign the Metastore** to one or more Databricks workspaces. * This links Unity Catalog governance across environments (e.g., dev, prod). * You can only assign **one Metastore per workspace**.   **(Optional) Create External Locations**   * If you want to manage access to external cloud storage (like raw data buckets), define **External Locations** and **Storage Credentials**. * Register them so Unity Catalog can track and secure data there.   **Initialize Unity Catalog Objects**   * In a workspace connected to the Metastore, create:   + **Catalogs** (e.g., sales\_data)   + **Schemas** (e.g., raw, curated)   + **Tables** (managed or external) * Use SQL or Terraform to define these resources.   **Grant Privileges**   * Assign permissions to **groups or service principals** for catalogs, schemas, and tables. * Example:   sql  CopyEdit  GRANT USAGE ON CATALOG sales\_data TO `sales\_analyst\_group`;  GRANT SELECT ON SCHEMA sales\_data.curated TO `bi\_consumers`;  **Validate the Setup**   * Test:   + Table creation.   + Access controls.   + Data lineage.   + Credential passthrough (if enabled).   **Automate (Recommended Best Practice)**   * Use **Terraform** or **Databricks CLI** to codify:   + Metastore creation.   + Storage credentials.   + External locations.   + Catalog and schema definitions.   + Permissions.   **Summary** These steps ensure you have:   * A **single, authoritative metadata layer**. * **Consistent access policies** across workspaces. * Clear **data lineage and governance**. |

Configure Clusters

|  |
| --- |
|  |
| **🟢 Steps to Set Up Clusters**  **Define Cluster Requirements**   * Decide what workloads you need to run:   + Interactive data exploration   + Scheduled ETL pipelines   + Machine learning training * Determine:   + Node types and sizes   + Autoscaling policies   + Pooling requirements for faster startup   **Choose Cluster Mode**   * **Single Node:** lightweight development and testing * **Standard Cluster:** multi-node processing * **High Concurrency Cluster:** optimized for serving many concurrent SQL queries   **Enable Unity Catalog Support**   * In cluster configuration:   + Enable **Unity Catalog access mode** (Single User or Shared Access)   + This ensures the cluster enforces Unity Catalog permissions and lineage tracking   **Configure Permissions**   * Set who can **attach notebooks** and **restart clusters** * Use cluster policies to standardize configurations (e.g., restrict node types or auto-termination settings)   **(Optional) Enable Credential Passthrough**   * If you need to access cloud storage using users’ identities, enable **credential passthrough** * Example: Azure Data Lake or AWS S3 with fine-grained permissions   **Provision the Cluster**   * Use:   + **Databricks UI**: Create manually via the Clusters page   + **Terraform**: Automate creation and ensure configuration consistency   + **REST API**: Programmatically spin up clusters as part of CI/CD pipelines   **Validate**   * Attach a notebook and test:   + Reading and writing Unity Catalog tables   + Credential passthrough access to external storage   + Performance and autoscaling behavior   **Automate Best Practices**   * Create **cluster policies** to enforce:   + Unity Catalog support   + Security hardening (e.g., table ACLs, credential passthrough)   + Tagging for cost tracking * Use **Terraform or the Databricks CLI** to declare cluster configurations in code   **Summary** These steps ensure your clusters are:   * **Properly governed under Unity Catalog** * Secure, scalable, and optimized for your workload patterns * Consistently deployed across environments |
| **🟢 Terraform Example for a Unity Catalog Cluster**  Below is an **example Terraform configuration** to create a **Single User cluster with Unity Catalog support** on Databricks:  hcl  CopyEdit  provider "databricks" {  host = "https://<databricks-instance>"  token = "<databricks-pat>"  }  resource "databricks\_cluster" "unity\_catalog\_cluster" {  cluster\_name = "unity-catalog-etl-cluster"  spark\_version = "14.3.x-scala2.12" # adjust based on latest supported versions  node\_type\_id = "Standard\_DS3\_v2" # example Azure node type  autotermination\_minutes = 60  autoscale {  min\_workers = 2  max\_workers = 8  }  data\_security\_mode = "SINGLE\_USER" # or "USER\_ISOLATION" if needed  # Attach to a specific instance profile (AWS) or managed identity (Azure) for credential passthrough  aws\_attributes {  instance\_profile\_arn = "arn:aws:iam::123456789012:instance-profile/databricks-unity-profile"  }  spark\_conf = {  "spark.databricks.repl.allowedLanguages" = "python,sql"  }  custom\_tags = {  Environment = "Production"  Owner = "data-eng-team"  }  library {  pypi {  package = "pandas"  }  }  }  **Key points:**   * data\_security\_mode = "SINGLE\_USER" **must be set for Unity Catalog** (this enables correct isolation). * For Azure, use azure\_attributes and configure your managed identity instead of aws\_attributes. * Always set autotermination\_minutes to control costs.   **🟢Cluster Policy Template Example**  A **Cluster Policy** ensures users can only create clusters with safe, compliant settings.  Here’s a **JSON template** to enforce Unity Catalog-compatible clusters:  json  CopyEdit  {  "policy\_family\_definition\_overrides": {  "data\_security\_mode": {  "value": "SINGLE\_USER",  "hidden": true  }  },  "definition": {  "spark\_version": {  "type": "fixed",  "value": "14.3.x-scala2.12"  },  "node\_type\_id": {  "type": "fixed",  "value": "Standard\_DS3\_v2"  },  "num\_workers": {  "type": "range",  "minValue": 2,  "maxValue": 8,  "defaultValue": 2  },  "autotermination\_minutes": {  "type": "fixed",  "value": 60  }  },  "description": "Policy to enforce Unity Catalog Single User clusters for secure data processing."  }  **How to apply:**   1. In Databricks, go to **Compute > Policies > Create**. 2. Paste this JSON. 3. Give the policy a name like:   pgsql  CopyEdit  Unity Catalog - Single User Policy   1. Save. 2. When users create clusters, have them **select this policy**.   **🟢Extra: Terraform Cluster Policy Resource**  If you want to **declare the policy in Terraform**, here’s a complete example:  hcl  CopyEdit  resource "databricks\_cluster\_policy" "unity\_catalog\_policy" {  name = "Unity Catalog Single User Policy"  definition = jsonencode({  "spark\_version": {  "type": "fixed",  "value": "14.3.x-scala2.12"  },  "node\_type\_id": {  "type": "fixed",  "value": "Standard\_DS3\_v2"  },  "num\_workers": {  "type": "range",  "minValue": 2,  "maxValue": 8,  "defaultValue": 2  },  "autotermination\_minutes": {  "type": "fixed",  "value": 60  },  "data\_security\_mode": {  "type": "fixed",  "value": "SINGLE\_USER"  }  })  }  **Tip:**   * You can assign this policy as default for all clusters or specific users/groups. |

Detailed Workflow

**Databricks End-to-End Workflow with Code Samples**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Component** | **Role in Workflow** | **Sample Code / Command** |  |
| **1** | **Delta Lake** | **Store ingested raw data in Delta format with ACID transactions.** | **Python**  **df = spark.read.csv("abfss://raw@storageacct.dfs.core.windows.net/sales/\*.csv", header=True)**  **df.write.format("delta")**  **.save("/mnt/sales/raw/")** | **This just physically writes Delta files:**   * Parquet + transaction logs (\_delta\_log) to your mounted storage location. * No table has been registered yet. * No Unity Catalog metadata is created automatically. * No permissions are assigned.   **You have Delta Lake files, but Unity Catalog doesn’t know anything about them yet.** |
| **2** | **Unity Catalog** | **Create governed catalogs, schemas, and register Delta tables.** | **Sql**  **CREATE CATALOG sales\_data;**  **CREATE SCHEMA sales\_data.raw;**  **USE SCHEMA sales\_data.raw;**  **CREATE TABLE transactions**  **USING DELTA LOCATION '/mnt/sales/raw/';** | Why?  After you have saved the Delta data, you must **register it as a table** so Unity Catalog can:   * Track metadata (columns, schema, location). * Apply permissions. * Enable lineage. * Allow BI tools or Databricks SQL to query it. |
|  | 1****What This Code Does**** python  CopyEdit  df = spark.read.csv("abfss://raw@storageacct.dfs.core.windows.net/sales/\*.csv", header=True)  df.write.format("delta").save("/mnt/sales/raw/")  ✅ **This just physically writes Delta files:**   * Parquet + transaction logs (\_delta\_log) to your mounted storage location. * **No table has been registered yet.** * **No Unity Catalog metadata is created automatically.** * **No permissions are assigned.**   ✅ So at this point:   * You have **Delta Lake files**, but **Unity Catalog doesn’t know anything about them yet**.  2 ****What Happens Next in Unity Catalog**** After you have saved the Delta data, you must **register it as a table** so Unity Catalog can:   * Track metadata (columns, schema, location). * Apply permissions. * Enable lineage. * Allow BI tools or Databricks SQL to query it.   This is done explicitly by creating a **table in Unity Catalog**. 3****How You Register the Data****Option 1 — ****Create a Managed Table**** If you want Unity Catalog to move and manage the data in a catalog’s storage root, you can:  sql  CopyEdit  CREATE TABLE sales\_data.raw.transactions  AS SELECT \* FROM delta.`/mnt/sales/raw/`;  ✅ This moves the data into the managed storage root (and tracks it in Unity Catalog). Option 2 — ****Create an External Table**** If you want the data to **stay in your raw path** (external location), do this:  sql  CopyEdit  CREATE TABLE sales\_data.raw.transactions  USING DELTA  LOCATION '/mnt/sales/raw/';  ✅ This registers the table in Unity Catalog but keeps the data where you saved it.  4**After You Register the Table**  Once you run the CREATE TABLE, Unity Catalog: ✅ Records the **metadata** (table name, schema, storage path). ✅ **Applies permissions** from the catalog and schema (e.g., GRANT SELECT). ✅ Enables **data lineage** in Unity Catalog. ✅ Allows **Databricks SQL, notebooks, and jobs** to query the data securely. ✅ Supports **credential passthrough** if configured. 5 ****Recommended Best Practice**** After writing Delta files: 1 Register them in Unity Catalog as **external tables**. 2 A**ssign access privileges** (e.g., grant usage and select to analyst groups). 3 Use Unity Catalog to track lineage and auditing. 6****Quick Example Workflow**** Here’s what a **typical workflow** looks like in practice:  ✅ **Step 1 — Write Data**  python  CopyEdit  df.write.format("delta").save("/mnt/sales/raw/")  ✅ **Step 2 — Register Table**  sql  CopyEdit  CREATE TABLE sales\_data.raw.transactions  USING DELTA  LOCATION '/mnt/sales/raw/';  ✅ **Step 3 — Grant Permissions**  sql  CopyEdit  GRANT SELECT ON TABLE sales\_data.raw.transactions TO `analyst\_group`;  ✅ **Step 4 — Query in Databricks SQL**  sql  CopyEdit  SELECT COUNT(\*) FROM sales\_data.raw.transactions;  ✅ **Summary** **Nothing happens automatically in Unity Catalog just by saving the Delta files.** You must **explicitly register** them as a Unity Catalog table—**this is the critical step** to bring governance, permissions, and lineage into play.   * Show Terraform examples to define tables. * Create example notebooks for this workflow.   Just let me know! You said: ignore box , any emoji can generate the code again ChatGPT said: Absolutely—here’s the same example **without boxes**, clear and clean, with a few ✨ emojis to keep it visual:  **1️⃣ Python: Save Delta Files**  python  CopyEdit  # 🚀 Ingest CSV into Delta Lake  df = spark.read.csv("abfss://raw@storageacct.dfs.core.windows.net/sales/\*.csv", header=True)  df.write.format("delta").save("/mnt/sales/raw/")  **2️⃣ SQL: Register the External Table in Unity Catalog**  sql  CopyEdit  -- 🏷️ Create an external table pointing to your Delta files  CREATE TABLE sales\_data.raw.transactions  USING DELTA  LOCATION '/mnt/sales/raw/';  **3️⃣ SQL: Grant Permissions**  sql  CopyEdit  -- 🔐 Give analysts read access  GRANT SELECT ON TABLE sales\_data.raw.transactions TO `analyst\_group`;  **4️⃣ SQL: Query the Table**  sql  CopyEdit  -- 📊 Verify data  SELECT COUNT(\*) FROM sales\_data.raw.transactions;  ✅ **Tip:**   * After Step 1, your data **just exists in storage.** * After Step 2, Unity Catalog **tracks and secures** it. | | | |
|  |  |  |  |  |
|  | | | | |
| **3** | **Metastore** | **Maintain metadata about all tables and views centrally.** | ***(Created and assigned via Databricks Admin Console or Terraform)*** |  |
|  |  | Great—let’s pick up exactly **after** you’ve:  ✅ Written Delta files ✅ Registered them in Unity Catalog as external tables ✅ Granted privileges  …and you asked:  👉 **What is the next step in the Metastore?** ****Next Steps in the Metastore**** After registering and granting permissions, **the Metastore is already holding all metadata about your tables**:   * **Catalog:** e.g., sales\_data * **Schema:** e.g., raw * **Table name:** transactions * **Storage location:** /mnt/sales/raw/ * **Schema definition:** columns, data types * **Table properties:** created\_by, created\_at * **Access control lists:** which groups have what permissions   In other words, **the Metastore has everything it needs to manage governance**.  But in a real workflow, **you don’t just stop there**. The next best practices and operational steps typically are: 1Data Validation and Quality Checks Before any downstream consumption (reporting or ML), validate:   * Record counts * Data completeness * Schema conformance * No unexpected NULLs or duplicates   **Example:**  sql  CopyEdit  SELECT COUNT(\*) FROM sales\_data.raw.transactions;  …and run expectations using **Delta expectations** or **Great Expectations.** ✅ 2Create Curated Tables or Views Move data from **raw** into **curated** schemas for clean consumption:  sql  CopyEdit  CREATE TABLE sales\_data.curated.transactions\_clean  AS  SELECT \*  FROM sales\_data.raw.transactions  WHERE amount > 0;  or  sql  CopyEdit  CREATE VIEW sales\_data.curated.latest\_sales AS  SELECT \*  FROM sales\_data.raw.transactions  WHERE transaction\_date >= current\_date() - INTERVAL '30' DAY; ✅ 3Assign Additional Privileges Grant access to **curated** or **analytics** layers to BI and ML users:  sql  CopyEdit  GRANT SELECT ON TABLE sales\_data.curated.transactions\_clean TO `bi\_team`; ✅ 4 Enable Data Lineage and Auditing Unity Catalog automatically tracks:   * Who created the table * Who modified it * Where the data came from (lineage graph)   Youtube : https://youtu.be/mjGPetUAhgg  ✅ No extra action needed here if you use Unity Catalog, but **you should monitor audit logs** via the **Databricks System Tables or external SIEM**. 5 Schedule Jobs for Ongoing Updates Set up jobs or workflows to:   * Incrementally load new data into raw * Transform into curated * Refresh reporting views   Example (Python):  python  CopyEdit  dbutils.notebook.run("/ETL/UpdateCuratedTables", 600)   ✅ 6 (Optional) Register in Feature Store If you use ML, you can create **Feature Store tables** using the curated data.   ****In Short: What happens in the Metastore next?**** Nothing additional happens automatically. But operationally, you now proceed to: ✅ Validate data ✅ Transform/curate it ✅ Grant access to downstream consumers ✅ Schedule refresh workflows ✅ Monitor lineage and audit logs  ✅ **Need help with:**   * Data quality patterns? * Automating transformations? * Defining lineage practices? | | |
|  |  |  |  |  |
| **4** | **Identity & Access Management** | **Define fine-grained access controls for teams (e.g., Sales Analysts).** | **Sql**  **GRANT SELECT ON TABLE sales\_data.raw.transactions TO `sales\_analyst\_group`;** |  |
|  |  | Excellent—let’s zoom in on Identity and Access Management (IAM) in this context: 👉 After you finish those Metastore steps—validating, transforming, granting table-level access, scheduling refresh workflows, and setting up lineage—the next considerations around IAM are about ensuring ongoing operational security and governance as your usage grows. Below are clear next steps for IAM, building on your Metastore work: ChatGPT said: Absolutely—let’s focus on **Identity and Access Management (IAM)** next steps **after you’ve completed the core Metastore tasks** (validation, curation, access grants, scheduling refresh, lineage).  Here are **clear IAM next steps** to keep your environment secure and governable as adoption scales:  ✅ **1. Review and Refine Privilege Model**   * Audit which principals (users, service principals, groups) have which permissions. * Use **least privilege**: remove any broad grants (e.g., ALL PRIVILEGES on catalogs or schemas) unless strictly necessary. * For Unity Catalog: confirm grants are scoped at the right level—catalog, schema, table, and fine-grained controls.   ✅ **2. Enforce Authentication Standards**   * Require **Single Sign-On (SSO)** integration (e.g., Azure AD, Okta, Google). * Enforce **multi-factor authentication (MFA)** for interactive users. * Rotate service principal credentials or tokens regularly.   ✅ **3. Define Role Hierarchies and Group Membership**   * Create **logical roles aligned to responsibilities** (e.g., Data Steward, Data Engineer, Analyst). * Map users and service accounts to these roles through groups in your IdP (Identity Provider). * Avoid direct assignment of permissions to individual users whenever possible.   ✅ **4. Set Up Row and Column-Level Security**   * Implement **Row-Level Security (RLS)** policies for sensitive datasets. * Apply **Column Masking** or dynamic views to restrict PII and sensitive fields. * Test policy enforcement with different roles to validate access boundaries.   ✅ **5. Enable Audit Logging and Monitoring**   * Ensure **Unity Catalog audit logs** are enabled and integrated with your SIEM (Splunk, Sentinel, etc.). * Monitor access patterns for anomalies (e.g., unusual data export volumes). * Set up alerting on critical security events (failed login attempts, policy changes).   ✅ **6. Automate Permission Lifecycle Management**   * Use infrastructure-as-code (Terraform, Databricks CLI) to **codify grants and role assignments**. * Regularly run automated jobs to detect and remediate drift (permissions diverging from your baseline). * Implement periodic reviews (e.g., quarterly) to re-certify access.   ✅ **7. Secure Service Workflows and Jobs**   * Confirm that scheduled jobs, notebooks, and pipelines use **service principals** with scoped permissions instead of personal user credentials. * Rotate tokens and secret scopes tied to production workflows. * Validate that job clusters don’t have excessive permissions by default.   ✅ **8. Document and Train**   * Maintain a **living IAM policy document** describing roles, access models, and approval workflows. * Train data owners and engineers to request and grant access in a controlled, auditable way. * Clearly define who owns stewardship of each dataset and who can approve access.   These steps help you **mature your IAM posture beyond initial setup**, supporting secure operations, scalability, and compliance.  If you’d like, I can help you draft:   * An **IAM policy template** * A **role mapping matrix** * Sample **Terraform snippets** for automating permissions   Just let me know! | | |
|  |  |  |  |  |
| **5** | **External Data Sources & Locations** | **Register the cloud storage container as an external location for governed access.** | **Sql**  **CREATE STORAGE CREDENTIAL my\_credential WITH AZURE\_MANAGED\_IDENTITY 'my-managed-identity';**  **CREATE EXTERNAL LOCATION raw\_sales\_loc URL 'abfss://raw@storageacct.dfs.core.windows.net/sales/' WITH (STORAGE CREDENTIAL my\_credential);** |  |
|  |  | what the next role for EXternal source and location after • An IAM policy template • A role mapping matrix • Sample Terraform snippets for automating permissions ChatGPT said: Great—once you’ve: ✅ Created your **IAM policy template** ✅ Built a **role mapping matrix** ✅ Developed **Terraform or infrastructure-as-code automation**  …the next important step is to **secure and govern EXTERNAL SOURCES and LOCATIONS** (e.g., cloud storage buckets, external tables, partner data).  This phase is essential to protect your data perimeter and avoid accidental overexposure.  Here’s **what comes next in role definition and setup for external sources and locations**: 🎯 Next Role Focus for External Sources and Locations ✅ **1. External Location Admin / Storage Steward Role**   * **Purpose:**   + Manage registration of external locations (e.g., S3 buckets, ADLS containers, GCS buckets).   + Set storage credentials and control read/write scope. * **Typical Responsibilities:**   + Register external locations in Unity Catalog.   + Create and manage storage credentials securely (e.g., Azure Service Principal, AWS IAM Role).   + Audit usage and validate access paths. * **Examples of Role Names:**   + external\_location\_admin   + storage\_steward   + bucket\_owner   ✅ **2. External Table Owner / Data Ingestion Role**   * **Purpose:**   + Create, refresh, and manage external tables over registered locations. * **Typical Responsibilities:**   + Define external tables and schema mappings.   + Manage refresh schedules and metadata sync.   + Validate access to external datasets is compliant. * **Examples of Role Names:**   + external\_table\_owner   + data\_ingestion\_engineer   ✅ **3. Storage Credential Manager**   * **Purpose:**   + Govern the lifecycle of credentials that Unity Catalog uses to access storage. * **Typical Responsibilities:**   + Rotate credentials periodically.   + Approve or revoke credential use.   + Ensure encryption policies are enforced. * **Examples of Role Names:**   + storage\_credential\_admin   + external\_access\_manager   ✅ **4. External Access Auditor**   * **Purpose:**   + Monitor and audit all reads and writes to external sources. * **Typical Responsibilities:**   + Review audit logs for data movement events.   + Confirm only authorized principals are accessing data.   + Enforce compliance requirements (e.g., data residency, retention policies). * **Examples of Role Names:**   + external\_access\_auditor   + data\_compliance\_reviewer   ✅ **5. Optional: Partner Data Steward / Third-Party Data Owner**   * **Purpose:**   + Govern ingestion and sharing of datasets from external partners. * **Typical Responsibilities:**   + Maintain contracts and usage policies.   + Approve external sharing.   + Validate data quality and licensing. * **Examples of Role Names:**   + partner\_data\_steward   + third\_party\_data\_owner  🌐 What You Should Do Next Here’s a **step-by-step flow after your IAM templates and automation are ready:**  1️⃣ **Define these external roles clearly.** 2️⃣ **Map who should own each role (teams, service principals, individuals).** 3️⃣ **Codify permissions (Terraform or CLI) for:**   * CREATE EXTERNAL LOCATION * CREATE STORAGE CREDENTIAL * READ FILES and WRITE FILES on the locations 4️⃣ **Enforce access controls:** * Grant usage only to the correct principals. * Use fine-grained privileges (READ FILES, WRITE FILES, USE CREDENTIAL). 5️⃣ **Enable logging & monitoring of external access events.** | | |
|  |  |  |  |  |
| **6** | **Clusters** | **Run compute workloads for ingestion, transformation, and curation.** | ***(Created via Databricks UI, Terraform, or API—configured with Unity Catalog support)*** |  |
|  |  | Great—let’s walk through this **step by step** so you see exactly **what comes next for Clusters** after you’ve:   * Created your **Storage Credential** * Created your **External Location** * Read/written files * Enforced access controls * Enabled monitoring   Below is the **logical sequence of next steps you should take for Clusters** in a Unity Catalog setup:  **🎯 Next Steps for Clusters**  Once you have secured and configured your storage integrations, here’s what you do for Clusters:  **✅ 6Configure Clusters to Use Unity Catalog**  **What this means:** Clusters must be explicitly set to **support Unity Catalog** so that your external locations and access controls are enforced.  **How to do it:** When creating a cluster:   * **Enable Unity Catalog** in the cluster configuration. * Make sure the **access mode** is set to *Single User* or *Shared* (recommended for Unity Catalog). * **Attach the correct instance profile** (AWS) or **Managed Identity** (Azure) if needed for passthrough.   **Databricks UI example:** ✅ Cluster > Create Cluster >   * Access mode: **Shared** (or Single User) * Unity Catalog: **Enabled**   **✅ 7Attach the Cluster to the Correct Metastore**  Clusters must be attached to the Unity Catalog **metastore** you created.  **Why:**   * Ensures metadata and table permissions are enforced at runtime. * Allows cluster workloads to resolve storage credentials and external locations.   **✅ 8 Verify Credential Passthrough (if using)**  If you plan to leverage **credential passthrough** (e.g., Azure AD passthrough), validate that:   * The cluster identity (e.g., Managed Identity or instance profile) has been granted permissions on the storage account/container. * Passthrough is enabled in cluster settings.   **✅ 9 Test Data Access with Cluster**  **Why:**   * You must validate that your cluster **can read and write only the expected data**. * Use test queries and write operations to confirm access controls.   **Example Test:**  sql  CopyEdit  -- Read test  SELECT \* FROM delta.`abfss://raw@yourstorage.dfs.core.windows.net/sales/`;  -- Write test  CREATE TABLE sales\_data.raw.sample\_test (id INT) LOCATION 'abfss://raw@yourstorage/sales/sample\_test/';  **✅ 10 Apply Cluster Policies**  **Why:**   * Cluster policies ensure **standardized configuration** across teams. * Prevent users from accidentally disabling Unity Catalog or using incorrect access modes.   **How:**   * Create Cluster Policies (e.g., require Unity Catalog enabled, enforce credential passthrough). * Assign policies to groups or workspaces.   ✅ **Summary Table of Next Steps**   |  |  |  | | --- | --- | --- | | **Step** | **Action** | **Purpose** | | **6** | Configure Cluster with Unity Catalog | Enable governance on all compute workloads. | | **7** | Attach Cluster to the Metastore | Ensure catalog metadata and permissions are enforced. | | **8** | Verify Credential Passthrough | Confirm storage authentication is working correctly. | | **9** | Test Read/Write Access | Validate enforcement of fine-grained permissions. | | **10** | Apply Cluster Policies | Standardize and secure cluster configurations across teams. |   ✅ **Pro Tip:**   * **Never** let users create clusters without policies in production. * Use **Terraform or Databricks REST API** to automate cluster creation and enforce configuration. | | |
|  |  |  | | |
|  |  |  |  |  |
| **7** | **Databricks SQL** | **Query curated data for reporting and dashboards.** | **Sql**  **SELECT customer\_id, SUM(amount) AS total\_sales**  **FROM**  **sales\_data.raw.transactions**  **GROUP BY customer\_id;** |  |
|  |  | ✅ Excellent—let’s walk step by step so you see the **logical flow after Cluster setup and policy enforcement.**  **🎯 Next Steps for Databricks SQL**  Once you have:  ✅ Created storage credentials and external locations ✅ Enforced access controls and logging ✅ Configured Clusters with Unity Catalog ✅ Applied Cluster Policies  …it’s time to focus on **Databricks SQL** (the serverless / SQL warehouse side) so your business users, analysts, and BI tools can securely query the curated data.  Below are the **next steps in order of priority:**  **✅ 1Create SQL Warehouses**  **Why:** SQL Warehouses are compute endpoints for:   * Running SQL queries interactively * Powering BI dashboards (Tableau, Power BI, Looker) * Sharing curated datasets   **How:**   * Go to **SQL > SQL Warehouses** in the Databricks UI. * Create new warehouses with:   + **Unity Catalog enabled** (critical)   + Proper **instance size** and scaling configuration   + Access mode set to **Shared** (recommended)   **Tip:** You can create multiple warehouses for different workloads (e.g., *reporting*, *ad hoc*, *data science*).  **✅ 2 Assign Permissions on Warehouses**  **Why:** Not all users should be able to start or stop SQL Warehouses. **How:** Grant USE WAREHOUSE to the right groups only. **Example SQL:**  sql  CopyEdit  GRANT USE WAREHOUSE ON WAREHOUSE finance\_reporting TO `finance\_analysts`;  **✅ 3Validate Warehouse Access to Unity Catalog Tables**  **Why:** You need to confirm that the warehouse can read governed data. **How:** Run test queries:  sql  CopyEdit  SELECT \* FROM sales\_data.curated.transactions LIMIT 10;  ✅ Validate:   * No permission errors * Correct data visibility   **✅ 4Configure BI Integrations**  **Why:** Most organizations use BI tools to consume curated data. **How:**   * Copy the **JDBC/ODBC connection string** from your SQL Warehouse. * Configure credentials or SSO in the BI platform. * Optionally, create service principals for BI connections.   **✅ 5Set Query Governance and Limits**  **Why:** Prevent runaway queries and control costs. **How:**   * Enable **query history logging**. * Configure:   + Max query duration   + Concurrency limits   + Auto-stop after idle period   **✅ 6Monitor Query Performance**  **Why:** Ensure workloads are healthy and performant. **How:**   * Use the **Query History** UI. * Review:   + Slow queries   + Frequent timeouts   + User activity   **✅ 7 Educate Analysts on Naming Conventions**  **Why:** Avoid confusion about where data lives. **How:**   * Train them to reference fully-qualified names:   php-template  CopyEdit  <catalog>.<schema>.<table>  e.g., sales\_data.curated.transactions  ✅ **Summary Table: Next Steps for Databricks SQL**   |  |  |  | | --- | --- | --- | | **Step** | **Action** | **Purpose** | | **1** | Create SQL Warehouses | Provision compute endpoints for querying and BI consumption. | | **2** | Assign Warehouse Permissions | Control who can start/stop/query warehouses. | | **3** | Validate Access to Unity Catalog Tables | Ensure governed data is available and secured. | | **4** | Configure BI Integrations | Enable dashboards and reporting workflows. | | **5** | Set Query Governance and Limits | Control query cost and performance. | | **6** | Monitor Query Performance | Troubleshoot and optimize workloads. | | **7** | Educate Analysts on Naming Conventions | Ensure clarity and consistency in data usage. |   ✅ **Pro Tip:** **Never** create SQL Warehouses without:   * Enabling Unity Catalog * Assigning clear permissions * Configuring auto-stop settings | | |
|  |  |  |  |  |
| **8** | **Workspace** | **Develop notebooks for ETL pipelines, curation, and validation.** | **Python**  **# Example notebook code to transform**  **datadf\_clean = df.filter("amount > 0")**  **df\_clean.write.format("delta").mode("overwrite")**  **.save("/mnt/sales/curated/")** |  |
|  |  | Excellent—let’s continue the **logical sequence** of your deployment.  ✅ So far you have: 1**Configured Clusters** 2**Applied Cluster Policies** 3**Provisioned SQL Warehouses** 4**Validated permissions and BI access**  👉 **The next focus area is the Workspace itself.**  **🎯 Next Steps for Workspace After SQL Warehouses**  **Why this matters:** The **Workspace** is where users:   * Build notebooks * Share code * Run jobs * Collaborate …and it needs to be **secured, organized, and consistent.**   Below is a structured list of steps:  **✅ 1Organize Repos and Notebooks**  **Why:** You don’t want notebooks spread across random folders.  **How:**   * Establish a **standard folder structure**, e.g., * Use **Repos** (Git-backed) for production code.   **Tip:** Encourage teams to sync notebooks to Git (GitHub, Azure DevOps).  **✅ 2 Set Workspace Access Controls**  **Why:** You must define **who can see and manage assets**.  **How:**   * Use **Workspace Access Control Lists (ACLs)**:   + Grant CAN VIEW / CAN EDIT / CAN RUN   + Restrict sensitive folders * Example:   Analysts can **view** curated ETL notebooks but **not edit.**   * Assign permissions by **groups**, not individuals.   **✅ 3Configure Secrets**  **Why:** Secure credentials (e.g., API keys, database passwords) must **never be stored in plain text**.  **How:**   * Create a **Secret Scope**:   bash  CopyEdit  databricks secrets create-scope --scope my-scope   * Store secrets:   bash  CopyEdit  databricks secrets put --scope my-scope --key db-password   * Reference secrets in notebooks:   python  CopyEdit  dbutils.secrets.get(scope="my-scope", key="db-password")  **✅ 4Enable Workspace Audit Logging**  **Why:** You need a record of all user actions for compliance.  **How:**   * Enable **audit log delivery** to a cloud storage account. * Configure retention policies.   **✅ 5Apply Workspace-Level Policies**  **Why:** Control how the workspace is used.  **Examples:**   * Limit cluster creation to admins. * Require Unity Catalog enabled clusters. * Enforce credential passthrough.   **✅ 6Educate Users**  **Why:** Even the best controls fail if people don’t know how to use them.  **How:**   * Provide onboarding guides:   + Folder structure   + Naming conventions (<catalog>.<schema>.<table>)   + Secrets management   + Cluster usage policies   ✅ **Summary Table: Next Steps for Workspace**   |  |  |  | | --- | --- | --- | | **Step** | **Action** | **Purpose** | | **1** | Organize Repos and Notebooks | Ensure code is structured, versioned, and discoverable. | | **2** | Set Workspace Access Controls | Restrict visibility and editing to appropriate groups. | | **3** | Configure Secrets | Secure credentials and sensitive configuration. | | **4** | Enable Audit Logging | Track user and system activity for compliance. | | **5** | Apply Workspace Policies | Standardize and enforce governance rules. | | **6** | Educate Users | Train teams on best practices and proper usage. |   ✅ **Pro Tip:** Treat your Workspace as **production infrastructure**—don’t leave permissions open, and don’t skip audit logs.  ✅ **What comes after Workspace?** Typically:   * **Jobs orchestration** (automated pipelines) * **ML workflows** (Feature Store, model tracking) * Ongoing monitoring and optimization   If you’d like, I can help you detail those steps as well—just let me know!  Ask ChatGPT  Top of Form  Bottom of Form | | |
|  |  |  |  |  |
| **9** | **Jobs** | **Schedule ingestion and transformation pipelines automatically.** | **Python**  **dbutils.notebook.run("/ETL/CurateSalesData", 600)** |  |
|  |  | Great—this is exactly the right sequence to follow after **Workspace setup**.  Below, I’ll give you the **logical next steps broken down in 3 clear sections**, so you can see where each fits and what to prioritize: 🎯 ****Next Steps After Workspace****✅ 1Jobs Orchestration (Automated Pipelines) **Objective:** Operationalize ingestion, curation, and publishing of data in a **repeatable, scheduled, and auditable way**. 🔹 Steps: ✅ **Define Job Pipelines**   * Identify critical workflows (e.g., ingest raw files → curate Delta tables → publish curated tables). * Create Databricks Jobs referencing notebooks or Python scripts.   ✅ **Schedule Jobs**   * Configure periodic schedules (e.g., every 4 hours). * Use event triggers if needed (e.g., file arrival).   ✅ **Assign Permissions**   * Grant job permissions to service principals (automation identities) rather than human users. * Example:   bash  CopyEdit  databricks permissions jobs --job-id 123 --json '{  "access\_control\_list": [  {  "group\_name": "data\_engineering",  "permission\_level": "CAN\_VIEW"  }  ]  }'  ✅ **Enable Notifications**   * Configure email or webhook alerts on failure or success.   ✅ **Enforce Cluster Policies**   * Ensure jobs always run with Unity Catalog-enabled clusters.   ✅ **Document Pipelines**   * Maintain a clear mapping of:   + Inputs   + Transformations   + Outputs   + Schedule   + Owners  ✅ 2ML Workflows (Feature Store, Model Training, Model Serving) **Objective:** Enable **data scientists** to reliably create, register, and serve models using governed data. 🔹 Steps: ✅ **Create Feature Store Tables**   * Use databricks.feature\_store to define features.   python  CopyEdit  from databricks.feature\_store import FeatureStoreClient  fs = FeatureStoreClient()  fs.create\_table(  name="sales\_features",  primary\_keys=["customer\_id"],  schema=df.schema,  description="Features for sales prediction"  )  ✅ **Train and Log Models**   * Use MLflow for model tracking.   python  CopyEdit  import mlflow  mlflow.start\_run()  # Training code  mlflow.log\_metric("accuracy", 0.91)  mlflow.sklearn.log\_model(model, "model")  ✅ **Register Models**   * Move models to the **Model Registry** for versioning.   ✅ **Assign Access Controls**   * Restrict who can:   + Register models   + Promote to production   + Serve endpoints   ✅ **Deploy Model Serving**   * Configure real-time or batch inference.   ✅ **Monitor Model Performance**   * Set up alerts for model drift and prediction anomalies.  ✅ 3 Ongoing Monitoring and Optimization **Objective:** Ensure your environment stays healthy, performant, and compliant over time. 🔹 Steps: ✅ **Enable Audit Logs**   * Confirm audit logs are flowing to SIEM or logging workspace.   ✅ **Review Query and Job Metrics**   * Use the Query History and Job Run history dashboards.   ✅ **Optimize Table Performance**   * Apply Delta Lake optimizations:   sql  CopyEdit  OPTIMIZE sales\_data.curated.transactions ZORDER BY (customer\_id);  ✅ **Track Cost and Usage**   * Review cluster usage reports and warehouse utilization.   ✅ **Rotate Credentials**   * Periodically rotate storage credentials and secrets.   ✅ **Enforce Policy Updates**   * Review and update cluster policies and workspace controls.   ✅ **Educate Teams**   * Hold periodic training sessions for:   + Naming conventions   + Best practices   + New platform capabilities   ✅ **Summary Table of Next Steps**   |  |  | | --- | --- | | **Focus Area** | **Key Actions** | | **Jobs Orchestration** | Create jobs, schedule pipelines, assign permissions, set notifications, enforce cluster policies, document workflows. | | **ML Workflows** | Create Feature Store tables, train and register models, apply access controls, deploy serving endpoints, monitor model performance. | | **Monitoring & Optimization** | Enable audit logs, review query and job metrics, optimize Delta tables, track costs, rotate credentials, enforce updated policies, educate users regularly. |   ✅ **Pro Tip:** Treat this as **a lifecycle, not a one-time setup**—plan continuous improvements. | | |
|  |  |  |  |  |
| **10** | **Databricks Machine Learning** | **Use curated data for feature engineering and ML modeling.** | **Python**  **from databricks.feature\_store**  **import FeatureStoreClientfs = FeatureStoreClient()fs.create\_table(name="sales\_features", primary\_keys=["customer\_id"],**  **schema=df\_clean**  **.schema()**  **, description="Features for sales prediction")** |  |
|  | ✅ Great—this is exactly how you should think about **layering your architecture**:  **You’ve already:**   * **Ingested raw data** * **Transformed and curated it into clean Delta tables** * **Scheduled ETL pipelines**   **✅ Next: you want to use this curated data for Machine Learning.**  Below is a **structured roadmap** of what comes *immediately after* you have automated ingestion and curation pipelines:  **🎯 Next Steps for Databricks Machine Learning (ML)**  **✅ 1️⃣ Curate ML-Ready Datasets**  **Why:** Before you train models, your data must be:   * Clean * Feature-engineered * Versioned   **How:**   * Create additional transformations to prepare your dataset. * Store the curated dataset in a **dedicated schema** (ml\_ready) in Unity Catalog. * Example:   sql  CopyEdit  CREATE SCHEMA IF NOT EXISTS sales\_data.ml\_ready;  CREATE TABLE sales\_data.ml\_ready.training\_set AS  SELECT  customer\_id,  SUM(amount) AS total\_sales,  COUNT(\*) AS num\_purchases,  MAX(date) AS last\_purchase\_date  FROM sales\_data.curated.transactions  GROUP BY customer\_id;  **✅ 2Register Features in Feature Store**  **Why:** To reuse and track features consistently across models.  **How:** Use Databricks Feature Store:  python  CopyEdit  from databricks.feature\_store import FeatureStoreClient  fs = FeatureStoreClient()  fs.create\_table(  name="sales\_features",  primary\_keys=["customer\_id"],  schema=df.schema,  description="Features for sales prediction"  )  fs.write\_table(  name="sales\_features",  df=df,  mode="overwrite"  )  **✅ 3Train and Log Models with MLflow**  **Why:** MLflow ensures your experiments are reproducible, versioned, and tracked.  **How:**  python  CopyEdit  import mlflow  import mlflow.sklearn  with mlflow.start\_run():  model = train\_model(training\_data) # your training code  mlflow.sklearn.log\_model(model, "model")  mlflow.log\_metric("accuracy", accuracy)  **✅ 4 Register Models in Model Registry**  **Why:** Central place to manage versions and lifecycle stages.  **How:**  python  CopyEdit  result = mlflow.register\_model(  "runs:/<run-id>/model",  "SalesPredictionModel"  )  Then promote to staging/production via UI or API.  **✅ 5Assign Access Permissions on Models**  **Why:** To control who can transition, delete, or serve models.  **How:** In UI: **Model Registry > Permissions**  Or API:  python  CopyEdit  mlflow.set\_model\_permissions(  "SalesPredictionModel",  {"data\_scientists": "CAN\_EDIT"}  )  **✅ 6Deploy Model Serving Endpoints**  **Why:** Enable real-time scoring or batch scoring.  **How:** Use Databricks Model Serving:   * Go to **Serving** tab in Model Registry * Select model version * Click **Enable Serving**   **✅ 7 Monitor Model Performance**  **Why:** To track drift, accuracy, and inference volume.  **How:**   * Enable Model Monitoring * Log inference metrics to MLflow * Periodically retrain if drift is detected   ✅ **Summary Table: Next Steps for ML After Pipelines**   |  |  |  | | --- | --- | --- | | **Step** | **Action** | **Purpose** | | **1** | Curate ML-Ready Datasets | Prepare clean, feature-rich training data. | | **2** | Register Features in Feature Store | Standardize and version engineered features. | | **3** | Train and Log Models with MLflow | Track experiments and metrics. | | **4** | Register Models in Model Registry | Manage model versions and lifecycle stages. | | **5** | Assign Access Permissions | Control model governance and security. | | **6** | Deploy Model Serving Endpoints | Make predictions in real time or batch. | | **7** | Monitor Model Performance | Detect drift and maintain model accuracy over time. |   ✅ **Pro Tip:** **Always** use Unity Catalog tables for your ML training sets to maintain lineage and access control end-to-end. | | | |
|  |  |  |  |  |