Domain 1: Set up and configure an Azure Databricks environment

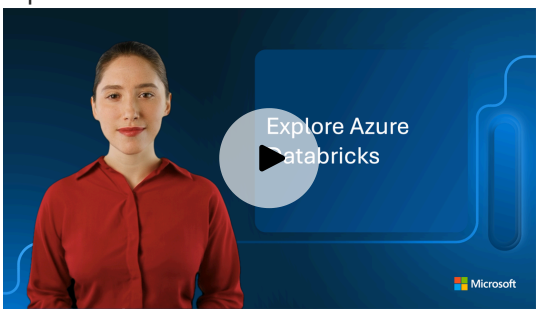
Explore Azure Databricks

1. Introduction

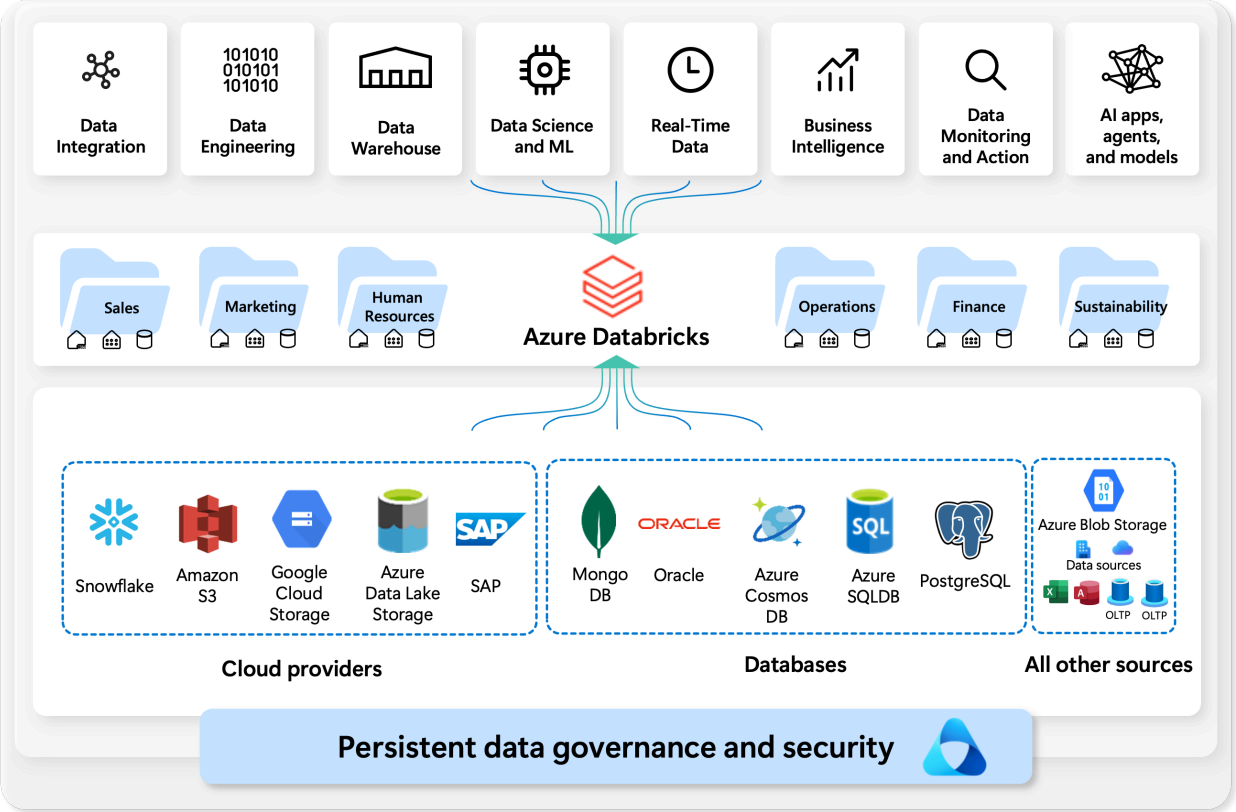
https://learn.microsoft.com/en-us/training/modules/explore-azure-databricks/01-introduction

Introduction

Completed



Azure Databricks is a cloud-based data platform that brings together the best of **data engineering, data science, and machine learning** in a single, unified workspace. Built on top of **Apache Spark**, it allows organizations to easily process, analyze, and visualize massive amounts of data in real time.



By connecting to a wide range of **data sources**—from cloud providers like Azure SQL Database, Amazon S3 and Google Cloud Storage, to enterprise systems such as SAP and Oracle—Azure Databricks makes it easy to integrate and transform data from anywhere.

Once data is ingested, teams across **sales, marketing, operations, finance, HR, and sustainability** can use Databricks for advanced analytics, machine learning, business intelligence, and AI-driven insights.

At its core, Azure Databricks helps organizations:

- **Integrate** data from multiple sources
- **Engineer** and transform raw data into usable formats
- **Store and manage** data efficiently with governance and security
- **Apply** real-time analytics, machine learning, and AI models
- **Drive** better business decisions and outcomes

Data Lakehouse

A **data lakehouse** is a data management approach that blends the strengths of both data lakes and data warehouses. It offers scalable storage and processing, allowing organizations to handle diverse workloads—such as machine learning and business intelligence—without relying on separate, disconnected systems. By centralizing data, a lakehouse supports a single source of truth, reduces duplicate costs, and ensures that information stays up to date.

Many lakehouses follow a layered design pattern where data is gradually improved, enriched, and refined as it moves through different stages of processing. This layered approach—commonly called the **medallion architecture**—organizes data into stages that build on one another, making it easier to manage and use effectively.

The Databricks lakehouse uses two key technologies:

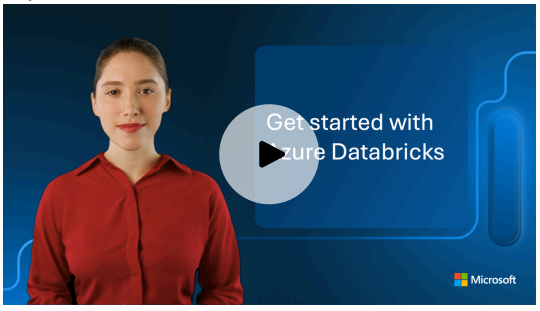
- **Delta Lake**: an optimized storage layer that supports ACID transactions and schema enforcement.
- **Unity Catalog**: a unified, fine-grained governance solution for data and AI.

2. Get started with Azure Databricks

https://learn.microsoft.com/en-us/training/modules/explore-azure-databricks/02-azure-databricks

Get started with Azure Databricks

Completed



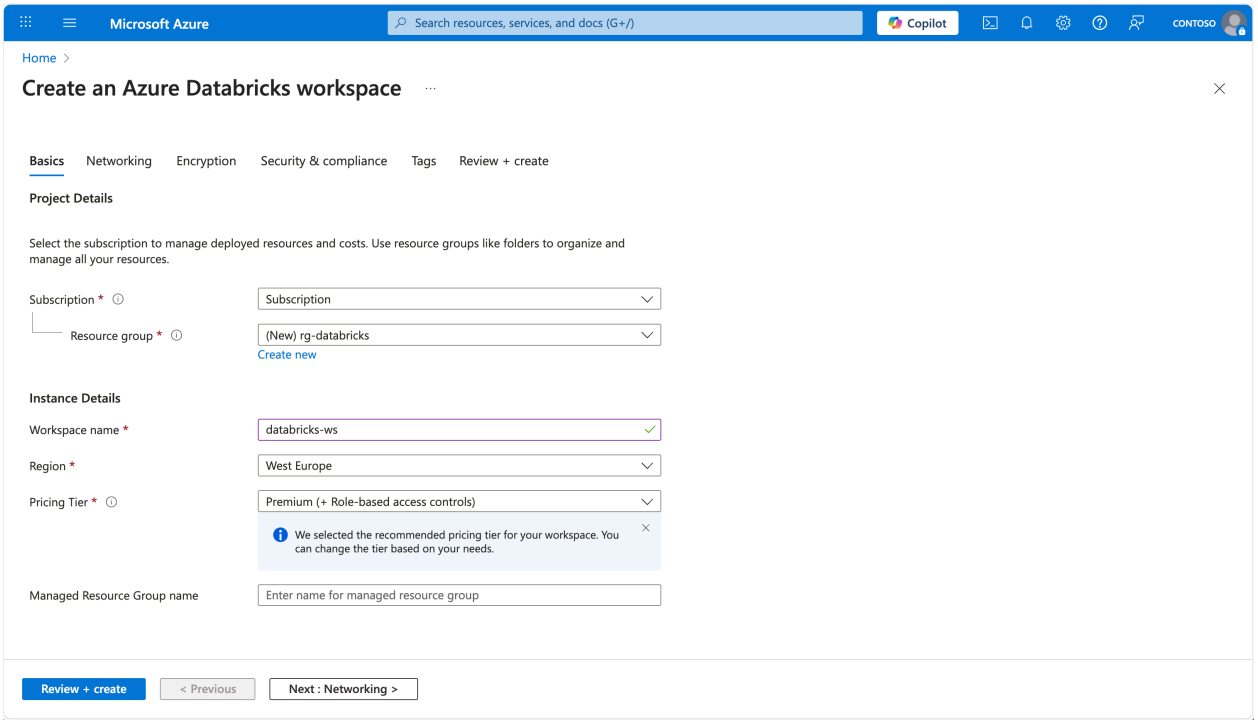
To use Azure Databricks, you must create an Azure Databricks workspace in your Azure subscription. A workspace is an Azure Databricks deployment in a cloud service account. It provides a unified environment for working with Azure Databricks assets for a specified set of users.

You can create an Azure Databricks workspace by:

- Using the Azure portal user interface.
- Using an Azure Resource Manager (ARM), Bicep or Terraform template.
- Using the New-AzDatabricksWorkspace Azure PowerShell cmdlet.
- Using the az databricks workspace create Azure command line interface (CLI) command.

When you create a workspace, you must specify:

- A **workspace name**.
- Select an available **region**. For available regions, see Azure services available by region.
- A **pricing tier**:
    - **Standard** - Core Apache Spark capabilities with Microsoft Entra ID integration.
    - **Premium** - Role-based access controls and other enterprise-level features.
    - **Trial** - A 14-day free trial of a premium-level workspace
- **Managed Resource Group name** (optional): an automatically created resource group where Azure provisions and manages the infrastructure resources needed for your Databricks workspace.



If you decide to create an Azure Databricks deployment using the Azure CLI, this would be the az databricks workspace command to remember:

```
az databricks workspace create
    --resource-group myresourcegroup \
    --name mydatabricksws  \
```

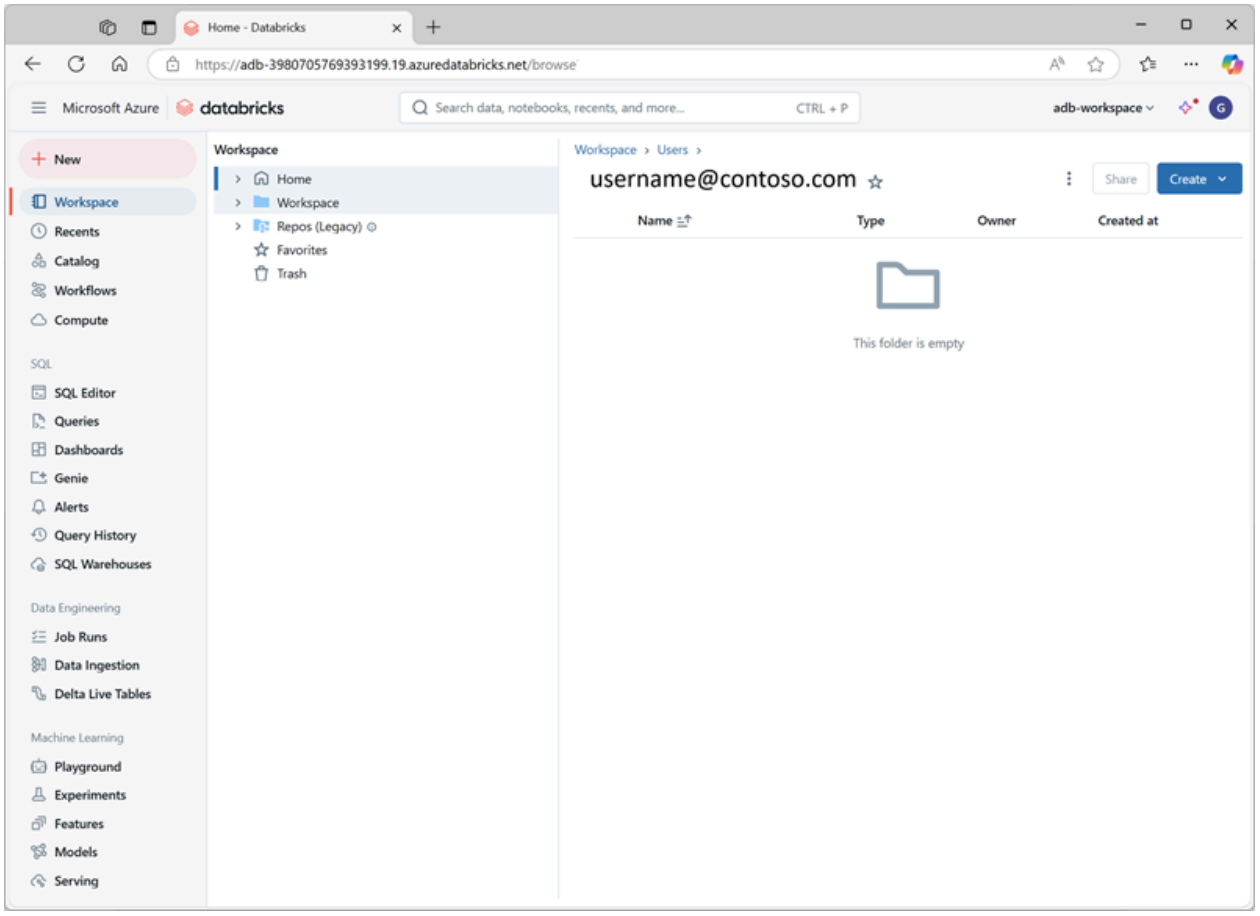
```
    --location westus2  \
    --sku standard
```

The equivalent New-AzDatabricksWorkspace PowerShell cmdlet:

```
New-AzDatabricksWorkspace -Name mydatabricksws -ResourceGroupName myresourcegroup -
```

Navigating the Azure Databricks Workspace UI

After you provision an Azure Databricks workspace, you can use the workspace UI to work with data and compute resources. The workspace UI is a web-based user interface where you can create and manage workspace resources, such as Spark clusters, and use notebooks and queries to work with data in files and tables.



The homepage provides shortcuts to common tasks and workspace objects to you help you get started. You can import data, create a notebook, create a query and configure an AutoML experiment.

The sidebar shows common Databricks categories (Workspace, Recents, Catalog, Jobs & Pipelines, Compute, Marketplace). It then breaks out by product area:

- **SQL**: SQL Editor, Queries, Dashboards, Genie, Alerts, Query History, SQL Warehouses
- **Data Engineering**: Job Runs, Data Ingestion

- **Machine Learning**: Playground, Experiments, Features, Models, Serving

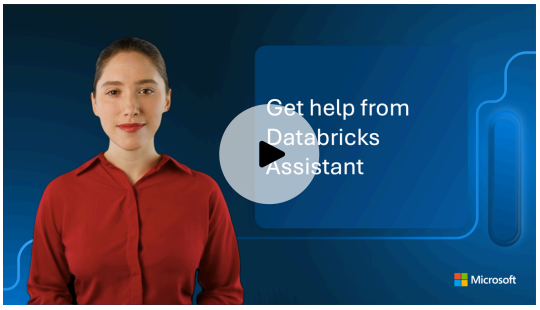
Select **+ New** to:

- **Create workspace objects** such as notebooks, queries, repos, dashboards, alerts, jobs, pipelines, experiments, models, and serving endpoints.
- **Create compute resources** such as clusters, SQL warehouses, and ML endpoints.

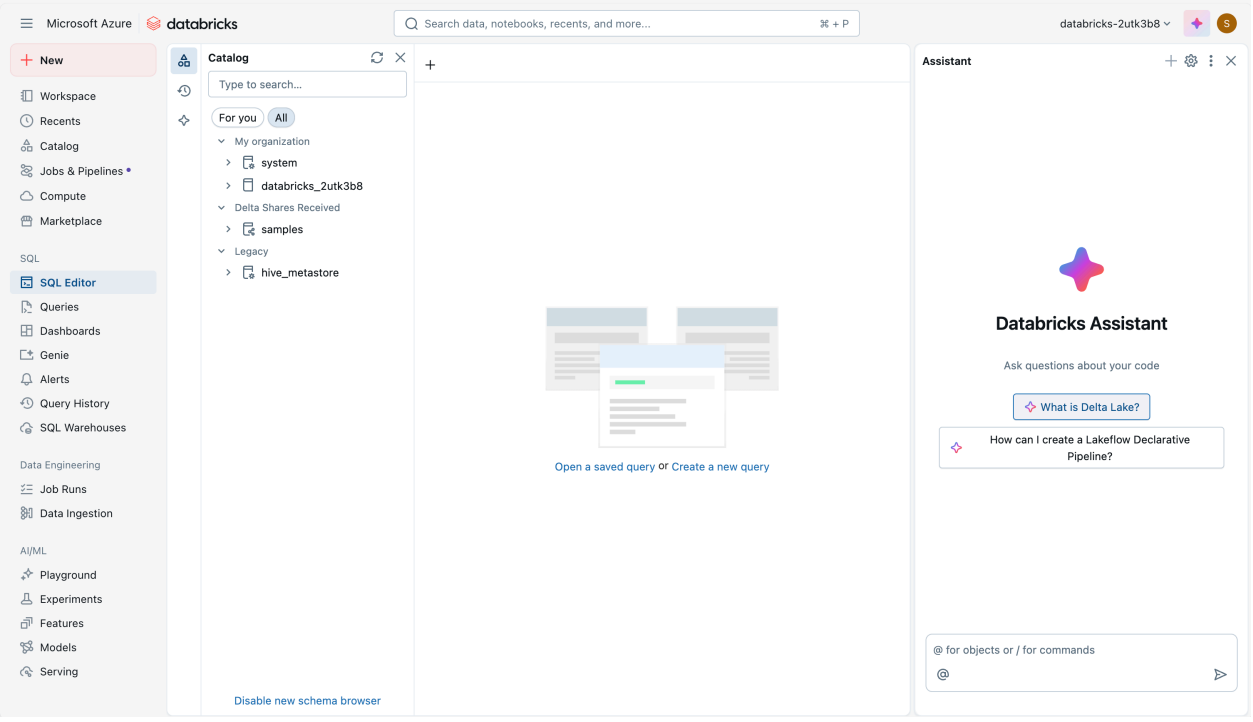
Use the top bar to **search** for workspace objects such as notebooks, queries, dashboards, alerts, files, folders, libraries, tables registered in Unity Catalog, jobs, and repos in a single place. You can also access recently viewed objects in the search bar.

The workspace is available in **multiple languages.** To change the workspace language, select your username in the top navigation bar, select **Settings** and go to the **Preferences** tab.

Get help from Databricks Assistant



Databricks Assistant is an AI-powered pair programmer and support tool that helps you work more efficiently in Databricks by generating, explaining, and fixing code or queries directly in notebooks, dashboards, and files.



It can assist with a wide range of tasks, including identifying and correcting errors, creating data visualizations, diagnosing job issues, and filtering or analyzing data using natural language prompts. The Assistant can surface relevant guidance from the Azure Databricks documentation.

By using Unity Catalog metadata, it personalizes its responses based on your organization's data assets—tables, columns, and descriptions—making it easier to explore and work with your data.

3. Identify Azure Databricks workloads

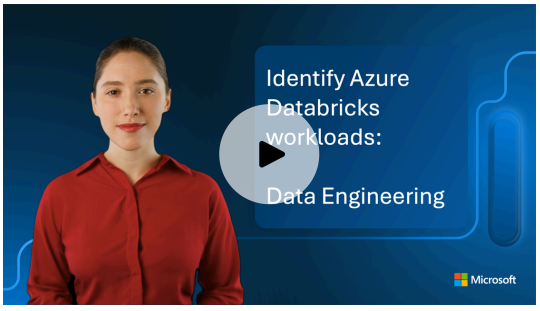
https://learn.microsoft.com/en-us/training/modules/explore-azure-databricks/03-workloads

Identify Azure Databricks workloads

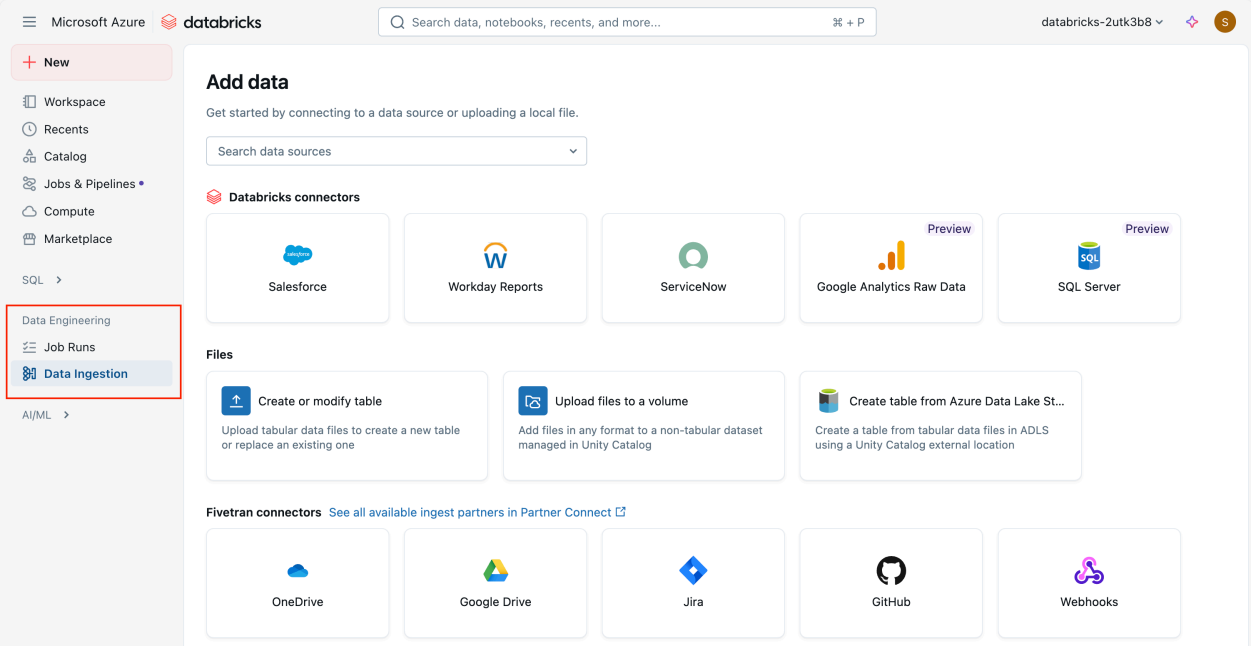
Completed

Azure Databricks offers capabilities for various workloads including Machine Learning and Large Language Models (LLM), Data Science, Data Engineering, BI and Data Warehousing, and Streaming Processing.

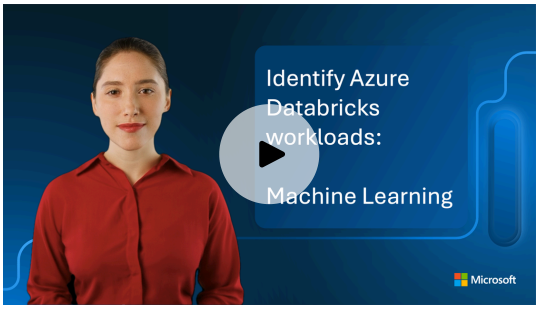
Data Engineering



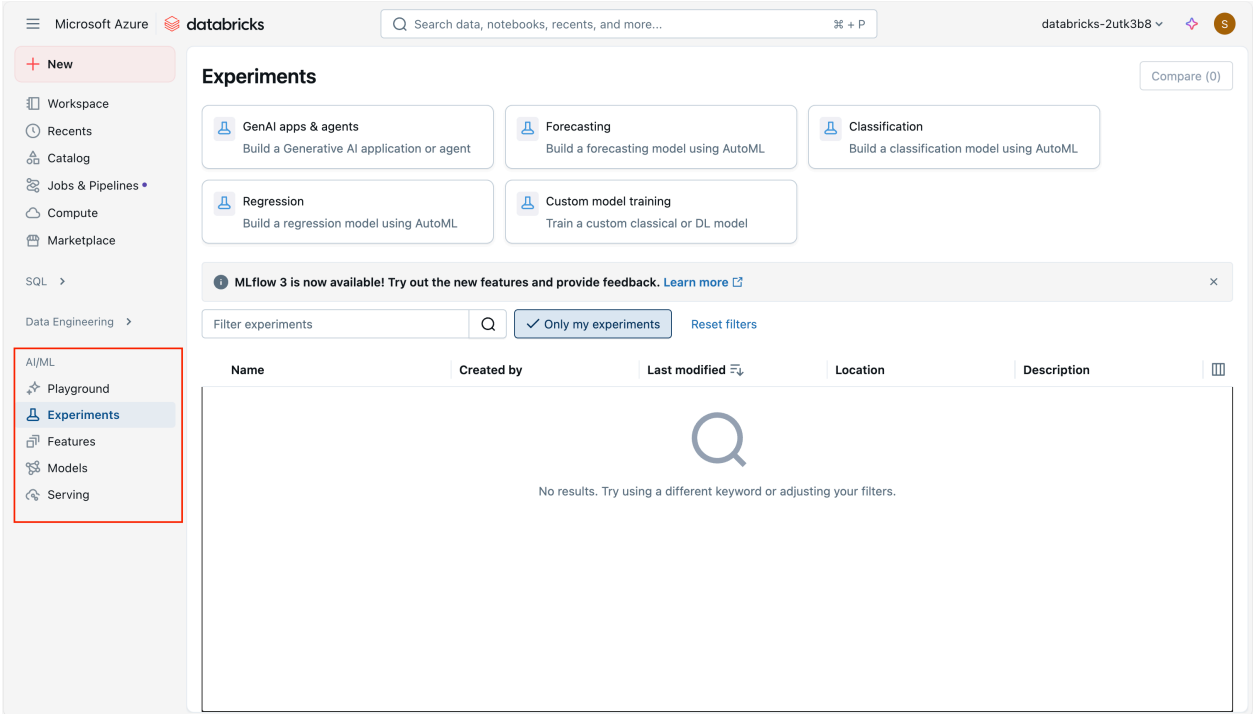
Azure Databricks provides capabilities for data scientists and engineers who need to collaborate on complex data processing tasks. It provides an integrated environment with Apache Spark for big data processing in a data lakehouse, and supports multiple languages including Python, R, Scala, and SQL. The platform facilitates data exploration, visualization, and the development of data pipelines.



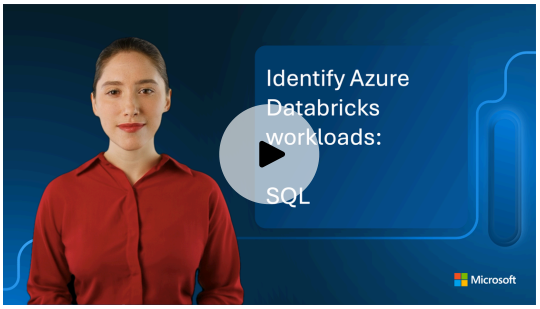
Machine Learning



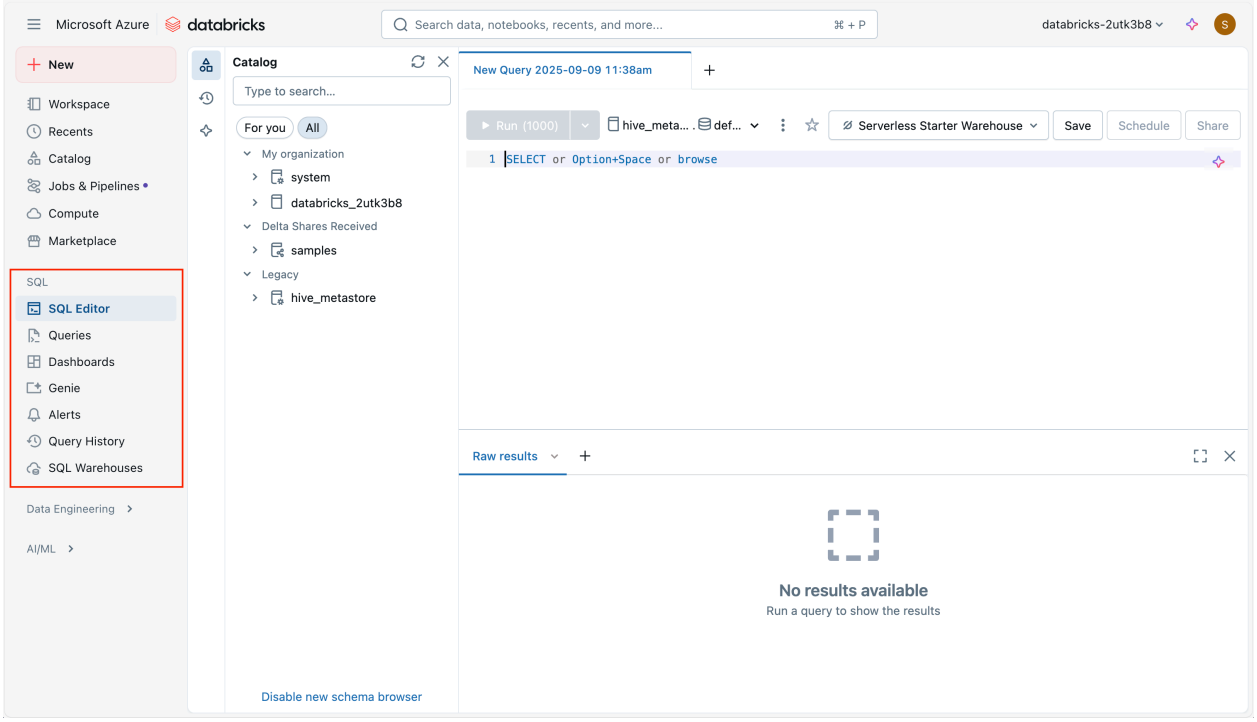
Azure Databricks supports building, training, and deploying machine learning models at scale. It includes MLflow, an open-source platform to manage the ML lifecycle, including experimentation, reproducibility, and deployment. It also supports various ML frameworks such as TensorFlow, PyTorch, and Scikit-learn, making it versatile for different ML tasks.



SQL



Data analysts who primarily interact with data through SQL can use SQL warehouses in Azure Databricks. The Azure Databricks Workspace UI provides a familiar SQL editor, dashboards, and automatic visualization tools to analyze and visualize data directly within Azure Databricks. This workload is ideal for running quick ad-hoc queries and creating reports from large datasets.



Note

SQL warehouses are included in the Premium (or higher) tier. Standard workspace doesn't provide SQL warehouses.

4. Understand key concepts

https://learn.microsoft.com/en-us/training/modules/explore-azure-databricks/04-key-concepts

Understand key concepts

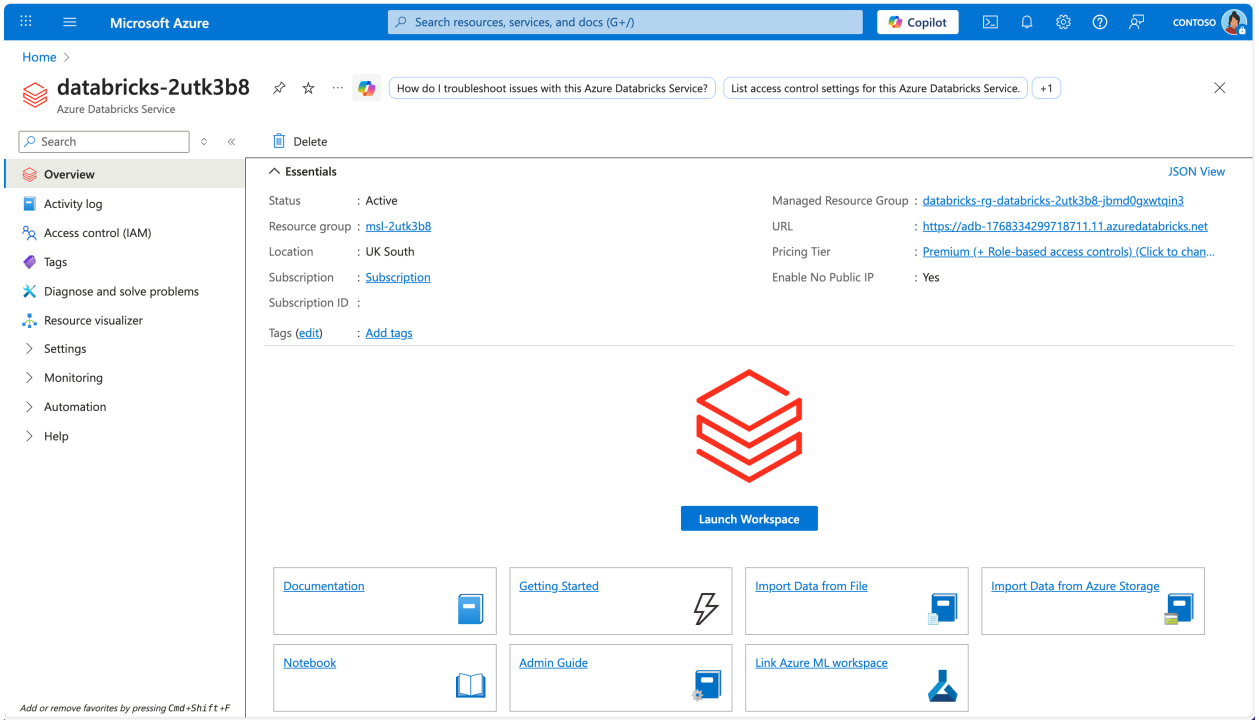
Completed

Azure Databricks is a single service platform with multiple technologies that enable working with data at scale. When using Azure Databricks, there are some key concepts to understand.

Workspaces

A **workspace** in Azure Databricks is a secure, collaborative environment where your can access and organize all Databricks assets, such as notebooks, clusters, jobs, libraries, dashboards, and experiments.

You can open an Azure Databricks Workspace from the Azure portal, by selecting **Launch Workspace**.



It provides a web-based **user interface (UI)** as well as REST APIs for managing resources and workflows. Workspaces can be structured into folders to organize projects, data pipelines, or team assets, and permissions can be applied at different levels to control access. They support **collaboration** by allowing multiple users—such as data engineers, analysts, and data scientists—to work together on shared notebooks, track experiments, and manage dependencies.

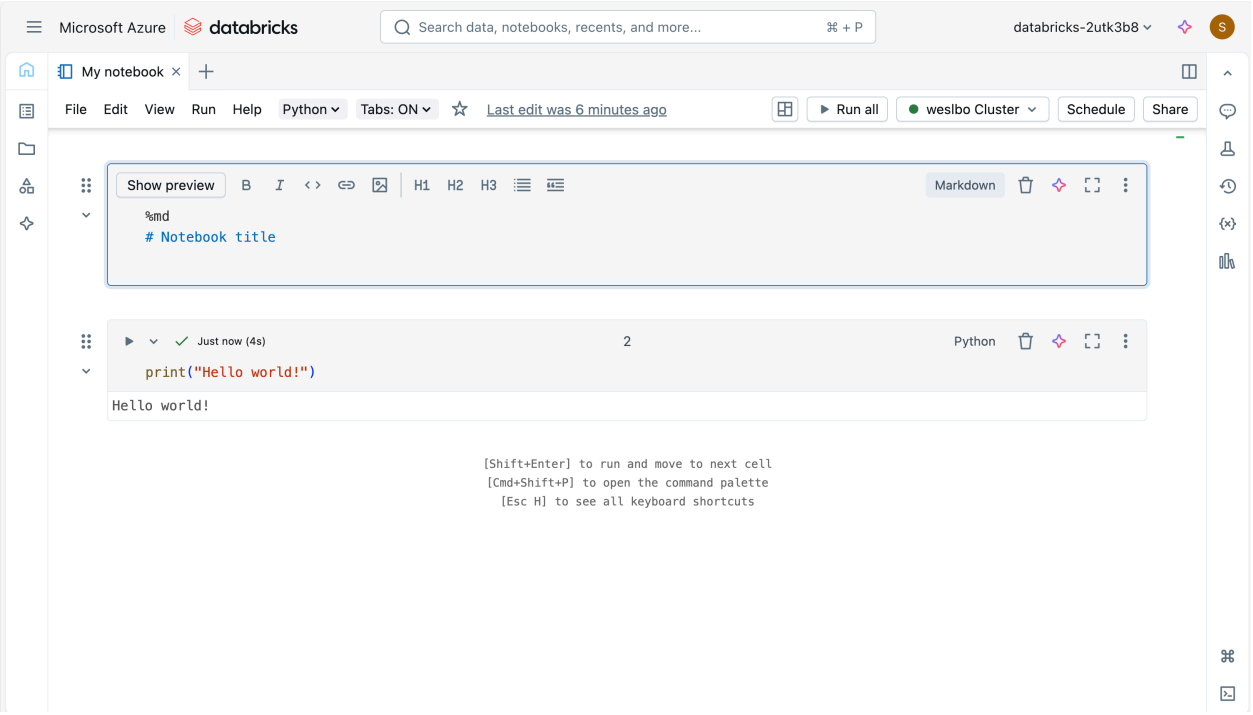
In addition, workspaces are tied to **Unity Catalog** (when enabled) for centralized data governance, ensuring secure access to data across the organization. Each workspace is also linked to an **underlying Azure resource group** (including a managed resource group) that holds the compute, networking, and storage resources Databricks uses behind the scenes.

Notebooks

Databricks notebooks are interactive, web-based documents that combine **runnable code, visualizations, and narrative text** in a single environment. They support multiple languages—such as Python, R, Scala, and SQL—and allow users to switch between languages within the same notebook using *magic commands*. This flexibility makes notebooks well-suited for **exploratory data analysis, data visualization, machine learning experiments, and building complex data pipelines**.

Notebooks are also designed for **collaboration**: multiple users can edit and run cells simultaneously, add comments, and share insights in real time. They integrate tightly with Databricks clusters, enabling users to process large datasets efficiently, and can connect to external data sources through **Unity Catalog** for governed data access. In addition, notebooks can be version-controlled, scheduled as jobs, or exported for sharing outside the platform, making them central to both **ad-hoc exploration** and **production-grade workflows**.

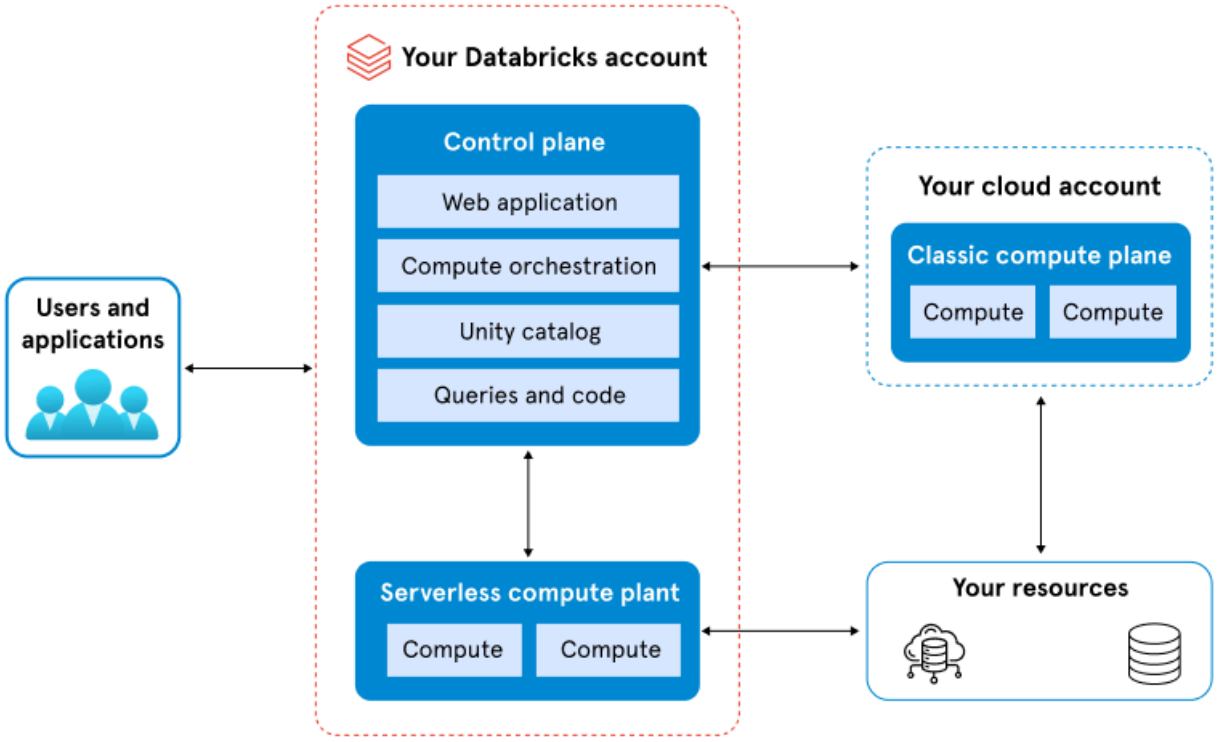
Notebooks contain a collection of two types of cells: **code cells** and **Markdown cells**. Code cells contain runnable code. Markdown cells contain Markdown code that renders as text and graphics. You can **run** a single cell, a group of cells, or the whole notebook.



Clusters

Azure Databricks leverages a two-layer architecture:

- **Control Plane**: this internal layer, managed by Microsoft, handles backend services specific to your Azure Databricks account.
- **Compute plane**: this is the external layer that processes the data and lives in your Azure Subscription.



Clusters are the core computational engines in Azure Databricks. They provide the processing power required to run data engineering, data science, and analytics tasks. Each cluster consists of a **driver node**, which coordinates execution, and one or more **worker nodes**, which handle the distributed computations. Clusters can be created manually with fixed resources or set to **auto-scale**, allowing Databricks to add or remove worker nodes depending on workload demand. This flexibility ensures efficient resource use and cost control.

Azure Databricks Compute offers a broad set of compute options available for different workload types:

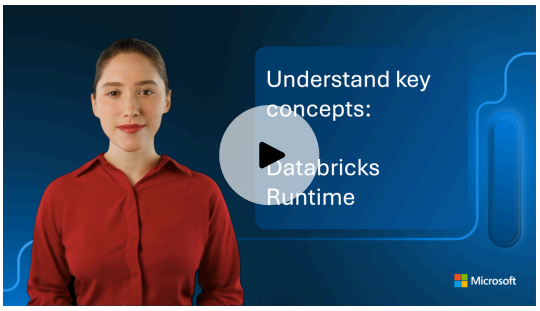
- **Serverless compute**: Fully managed, on-demand compute that automatically scales up or down to meet workload needs. Ideal for teams that want fast startup times, minimal

management overhead, and elastic scaling.

- **Classic compute**: User-provisioned and configured clusters that offer full control over compute settings, such as VM sizes, libraries, and runtime versions. Best for specialized workloads that require customization or consistent performance.
- **SQL warehouses**: Compute resources optimized for SQL-based analytics and BI queries. SQL warehouses can be provisioned as either **serverless** (elastic, managed) or **classic** (user-configured) depending on governance and performance requirements.

This allows you to tailor compute to specific needs—from exploratory analysis in notebooks, to large-scale ETL pipelines, to high-performance dashboards and reporting.

Databricks Runtime



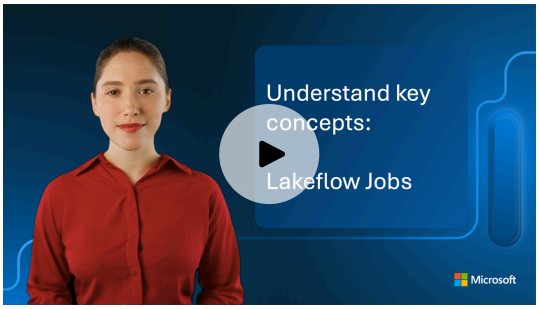
The **Databricks Runtime** is a set of customized builds of **Apache Spark** that include performance improvements and additional libraries. These runtimes make it easier to handle tasks such as **machine learning**, **graph processing**, and **genomics**, while still supporting general data processing and analytics.

Databricks provides multiple runtime versions, including **long-term support (LTS)** releases. Each release specifies the underlying Apache Spark version, its release date, and when support will end. Over time, older runtime versions follow a lifecycle:

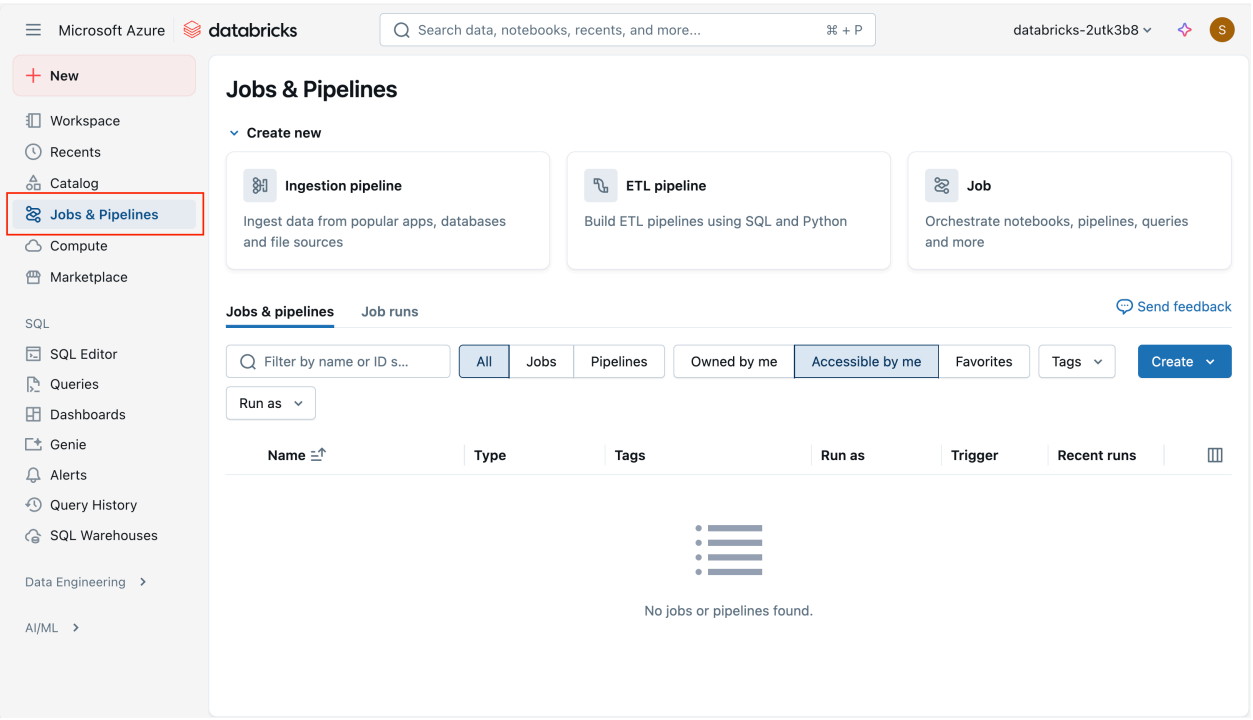
- **Legacy** – available but no longer recommended.
- **Deprecated** – marked for removal in a future release.
- **End of Support (EoS)** – no further patches or fixes are provided.
- **End of Life (EoL)** – retired and no longer available.

If a maintenance update is released for a runtime version you're using, you can apply it by **restarting your cluster**.

Lakeflow Jobs



Lakeflow Jobs provide workflow automation and orchestration in Azure Databricks, making it possible to reliably schedule, coordinate, and run data processing tasks. Instead of running code manually, you can use jobs to automate repetitive or production-grade workloads such as ETL pipelines, machine learning training, or dashboard refreshes.



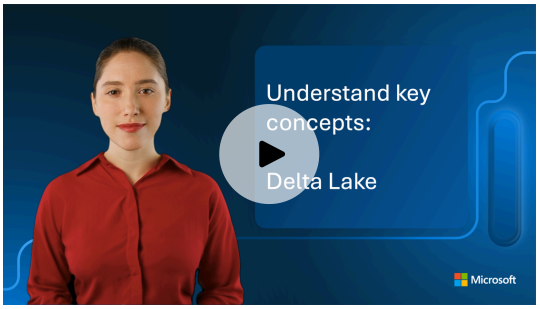
A job in Databricks is essentially a container for one or more **tasks**. Tasks define the work to be done—for example, running a notebook, executing a Spark job, calling external code, ...

Jobs can be triggered in different ways:

- On a schedule (for example, every night at midnight).
- In response to an event.
- Manually, when needed.

Because they're repeatable and managed, jobs are critical for **production workloads**. They ensure that data pipelines run consistently, ML models are trained and deployed in a controlled manner, and downstream systems receive updated, accurate data.

Delta Lake

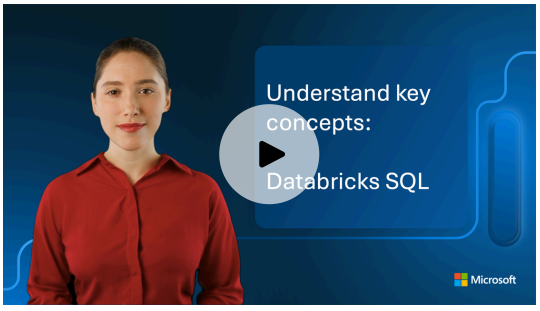


Delta Lake is an open-source storage framework that improves the reliability and scalability of data lakes by adding transactional features on top of cloud object storage, such as **Azure Data Lake Storage**. Traditional data lakes can suffer from issues like inconsistent data, partial writes, or difficulties managing concurrent access. Delta Lake addresses these problems by supporting:

- **ACID transactions** (atomicity, consistency, isolation, durability) for reliable reads and writes.
- **Scalable metadata handling** so tables can grow to billions of files without performance loss.
- **Data versioning and rollback**, enabling time travel queries and recovery of previous states.
- **Unified batch and streaming processing**, so the same table can handle real-time ingestion and historical batch loads.

On top of this foundation, **Delta tables** provide a familiar table abstraction that makes it easier to work with structured data using **SQL queries** or the **DataFrame API**. Delta tables are the **default table format in Azure Databricks**, ensuring that new data is stored with transactional guarantees by default.

Databricks SQL



Databricks SQL brings **data warehousing capabilities** to the Databricks Lakehouse, allowing analysts and business users to query and visualize data stored in open formats directly in the data lake. It supports **ANSI SQL**, so anyone familiar with SQL can run queries, build reports, and create dashboards without needing to learn new languages or tools.

Databricks SQL is available only in the **Premium tier** of Azure Databricks. It includes:

- A **SQL editor** for writing and running queries.
- **Dashboards and visualization tools** for sharing insights.
- Integration with external BI and analytics tools.

SQL Warehouses

All Databricks SQL queries run on **SQL warehouses** (formerly called SQL endpoints), which are scalable compute resources decoupled from storage. Different warehouse types are available depending on performance, cost, and management needs:

- **Serverless SQL Warehouses**
    - **Instant and elastic compute** with fast startup and autoscaling.
    - **Low management overhead** since Databricks handles capacity, patching, and optimization.
    - **Cost efficiency** by scaling automatically and avoiding idle resource costs.
- **Pro SQL Warehouses**
    - More customizable but slower to start (≈4 minutes).
    - Less responsive autoscaling compared to serverless.
    - Useful when consistent, predictable workloads are required.
- **Classic SQL Warehouses**
    - Compute resources run in your **own Azure subscription**, not in Databricks.
    - Less flexible than serverless but may be preferred for specific governance or cost management requirements.

MLflow

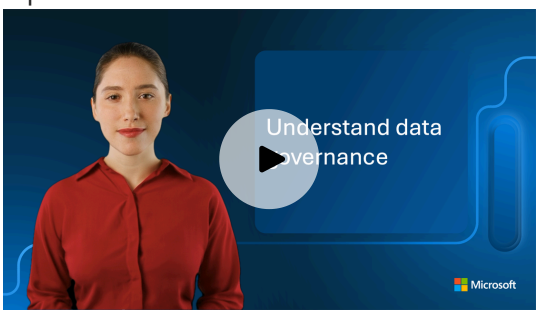
MLflow is an open-source platform designed to manage the **end-to-end machine learning (ML) lifecycle**. It helps data scientists and ML engineers track experiments, manage models, and streamline the process of moving models from development to production. MLflow also supports generative AI workflows and includes tools for evaluating and improving AI agents.

5. Data governance using Unity Catalog and Microsoft Purview

https://learn.microsoft.com/en-us/training/modules/explore-azure-databricks/05-data-governance-using-unity-catalog-and-microsoft-purview

Data governance using Unity Catalog and Microsoft Purview

Completed



Data governance is critical for ensuring that data within an organization is managed securely, efficiently, and in compliance with regulations.

In many organizations, data is distributed across databases, data warehouses, data lakes, and even multiple catalogs. It also exists in diverse formats like Parquet, CSV, and Delta Lake. Beyond structured data in tables, there's also unstructured data in files, along with other assets such as machine learning models, notebooks, and dashboards that require management and governance. This fragmentation creates silos across sources, formats, and asset types.

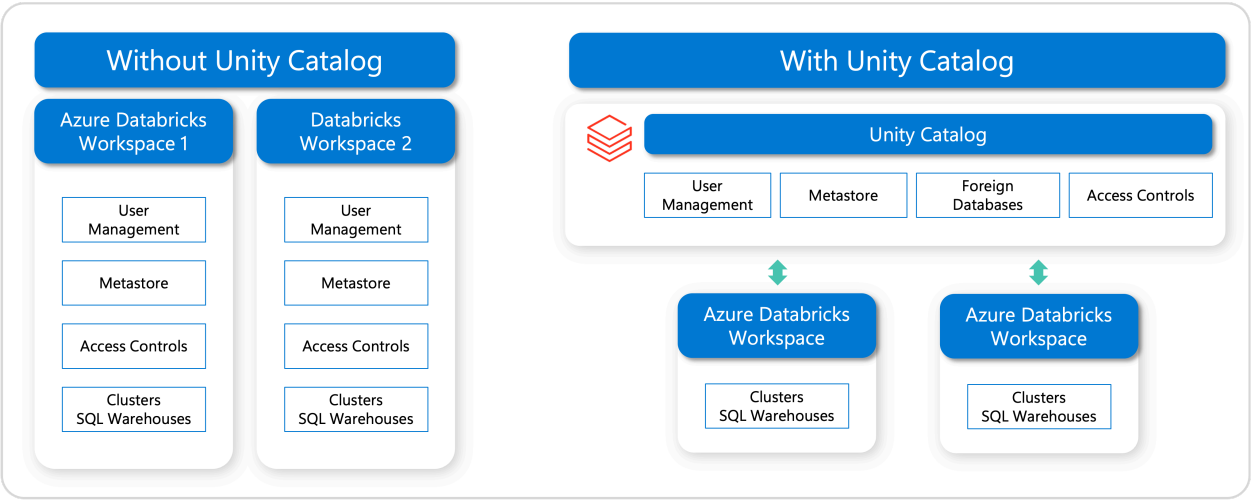
These governance challenges directly affect the value organizations can derive from data and AI:

- Fragmented governance increases compliance, security, and data quality risks, while also creating operational inefficiencies as teams struggle to maintain a consistent view of their data and AI environments.
- Limited connectivity can result in vendor lock-in and make it harder to adopt new technologies as requirements change. Poor interoperability also complicates collaboration and scaling, often leading to higher costs from using multiple tools and duplicating data across systems.
- A lack of built-in intelligence restricts broader use of data and AI platforms, particularly for nontechnical users. This slows down innovation, delays decision-making, and prevents organizations from fully realizing the benefits of their data and AI investments.

Azure Databricks, combined with Unity Catalog and Microsoft Purview, provides a robust solution for managing and governing data effectively.

Unity Catalog

Unity Catalog provides a centralized way to manage access, discovery, lineage, audit logs, and quality monitoring across data and AI assets within Azure Databricks. It applies consistently across all workspaces in a region.



The **metastore** is the top-level metadata container; it holds information about data assets and the permissions that govern them. You typically have one metastore per region, and multiple workspaces can share that metastore.

Unity Catalog organizes data assets using a structured **three-level hierarchy**:

```
catalog.schema.table_or_other_object
```

- **Catalogs** group assets typically aligned to teams or environments.
- **Schemas** (also known as databases) are subdivisions within catalogs, organizing assets more granularly—for example by project or use case.
- **Objects** in schemas include tables (managed or external), views, volumes, functions, and models.

Tables can be either managed or external. With **managed tables**, Unity Catalog handles both governance and storage (always Delta Lake format). With **external tables**, Unity Catalog manages access from Databricks, but data lifecycle/storage is managed externally. This supports multiple formats (Delta, CSV, JSON, Parquet, etc.)

Unity Catalog implements **fine-grained access control** via ANSI SQL commands across multiple levels—metastore, catalog, schema, down to rows and columns. For example, the following

command gives the 'finance-team' user group the permission to create new tables in the 'myschema' within the 'mycatalog' database.

```
GRANT CREATE TABLE ON SCHEMA mycatalog.myschema TO `finance-team`;
```

Exploring data assets in Unity Catalog is straightforward. You can use the **Catalog Explorer** and a search interface to find what you need. To help you, assets have tags, comments, and even AI-generated descriptions. Once you find a data asset, you can use features like **lineage**, table insights, and Entity Relationship diagrams to get a better understanding of it.

Unity Catalog provides a complete picture of your data's history. It logs access, audit trails, and lineage—right down to the column level.

In most accounts, Unity Catalog is enabled by default when you create a workspace. You can get started using Unity Catalog with the default settings. There are optional configurations that you might want to enable, however.

Microsoft Purview

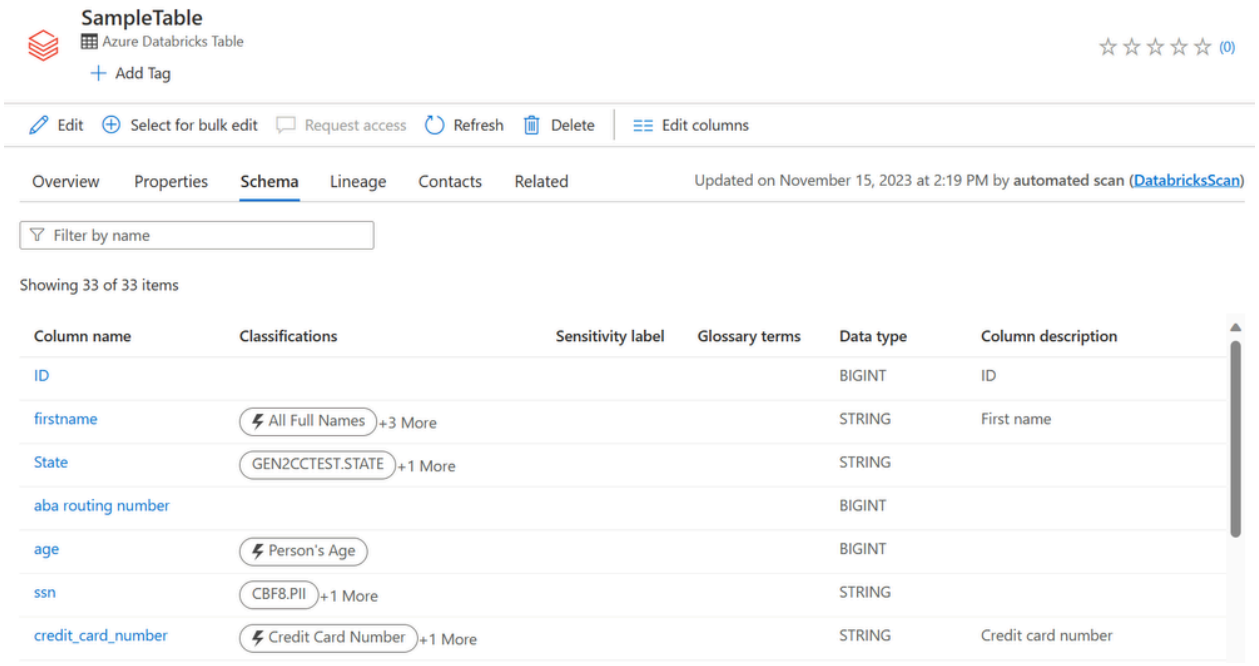
Microsoft Purview is a data governance service that lets you manage and oversee data across on-premises systems, multiple clouds, and SaaS platforms. It includes features such as data discovery, classification, lineage tracking, and access governance.

When integrated with Azure Databricks and Unity Catalog, Purview can discover Lakehouse data and ingest its metadata into the Data Map. This allows you to apply consistent governance across your entire data environment, while acting as a central catalog that brings together metadata from different sources.

With this integration, you can:

- **Scan** Azure Databricks in both public and private networks, powered by the fully managed Microsoft Purview integration runtime.
- Scan the entire **Unity Catalog metastore** or choose to scan only selective catalogs.
- Extract a comprehensive set of Unity Catalog metadata, including details of metastore, catalogs, schemas, tables/views, and columns, etc.
- Automatically **classify** the data based on built-in system classification rules or user-defined custom classification rules to identify sensitive data.

- Get detailed visibility into the **data lineage**, showing how data is transformed and moved across different systems and processes, including within Azure Databricks.
- Run the scan **on-demand** or on a daily/weekly/monthly recurring **schedule**.



In addition, Microsoft Purview can scan the workspace-level **Hive metastore** in Azure Databricks.

6. Exercise - Explore Azure Databricks

https://learn.microsoft.com/en-us/training/modules/explore-azure-databricks/06-exercise-explore-databricks

Exercise - Explore Azure Databricks

Completed

Now it's your chance to explore Azure Databricks. In this lab, you explore the Azure Databricks workspace UI, upload a sample dataset to a Unity Catalog volume, and work with notebook features including Python, SQL magic commands, and Markdown. You use the Databricks Assistant throughout to generate and refine code in the context of CityMoves Transit, a fictional public transportation authority.

> **Note**
>
> To complete this lab, you need an Azure subscription in which you have administrative access.

Launch the exercise and follow the instructions.

Launch Exercise

7. Module assessment

https://learn.microsoft.com/en-us/training/modules/explore-azure-databricks/07-knowledge-check

Module assessment

Completed

8. Summary

https://learn.microsoft.com/en-us/training/modules/explore-azure-databricks/08-summary

Summary

Completed

Azure Databricks is a cloud-based data analytics platform that provides a unified environment for data engineering, machine learning, and analytics. It's built on top of Apache Spark, which is a powerful, open-source processing engine built around speed, ease of use and sophisticated analytics. Azure Databricks integrates with other services provided by Microsoft Azure, offering a seamless experience for data preparation, building machine learning models, and data analysis.

In this module, you learned how to:

- Provision an Azure Databricks workspace
- Identify core workloads for Azure Databricks
- Use Data Governance tools Unity Catalog and Microsoft Purview
- Describe key concepts of an Azure Databricks solution

Learn more

- Azure Databricks Documentation

- Getting started with Azure Databricks
- Microsoft Purview Documentation
- Databricks Unity Catalog

Understand Azure Databricks architecture

1. Introduction

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-architecture/1-introduction

Introduction

Completed

When you deploy Azure Databricks for data engineering workloads, understanding its architecture becomes essential for making informed decisions about data organization, security configurations, and resource management. The architecture determines where your data resides, how compute resources are allocated, and how different components interact to provide the unified analytics platform you rely on.

Azure Databricks organizes resources through a hierarchical structure that spans from account-level governance down to individual data objects. At the foundation, the **control plane** manages orchestration and configuration while the **compute plane** processes your data—either in **serverless** environments fully managed by Azure Databricks or in **classic compute** running within your Azure subscription. This separation enables you to maintain security and governance while scaling workloads efficiently.

Storage options in Azure Databricks have evolved to meet different organizational needs. **Default storage** simplifies getting started by providing fully managed storage in serverless workspaces without configuration overhead. **External storage** connects Unity Catalog to your existing cloud storage accounts, enabling you to work with data managed outside Azure Databricks while maintaining governance. **Unity Catalog managed storage** lets you define where Unity Catalog stores data at the catalog or schema level, providing fine-grained control over data placement while Unity Catalog handles the lifecycle.

Throughout this module, you explore how these architectural components work together. You learn how the **account hierarchy** organizes resources across workspaces and metastores, how control and compute planes separate responsibilities, and how different storage patterns serve specific use cases. By understanding these fundamentals, you'll be equipped to configure Azure Databricks environments that align with your organization's security, governance, and operational requirements.

2. Understand Azure Databricks architecture

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-architecture/2-understand-azure-databricks-architecture

Understand Azure Databricks architecture

Completed

When you work with Azure Databricks as a data engineer, understanding its architecture helps you make informed decisions about how to organize your data, configure compute resources, and manage security. The architecture determines how your workloads run, where your data resides, and how different components interact.

Azure Databricks uses a hierarchical structure that organizes resources from the account level down to individual data objects. This structure, combined with the separation between control and compute planes, provides flexibility and security for your data engineering workloads.

Azure Databricks account hierarchy

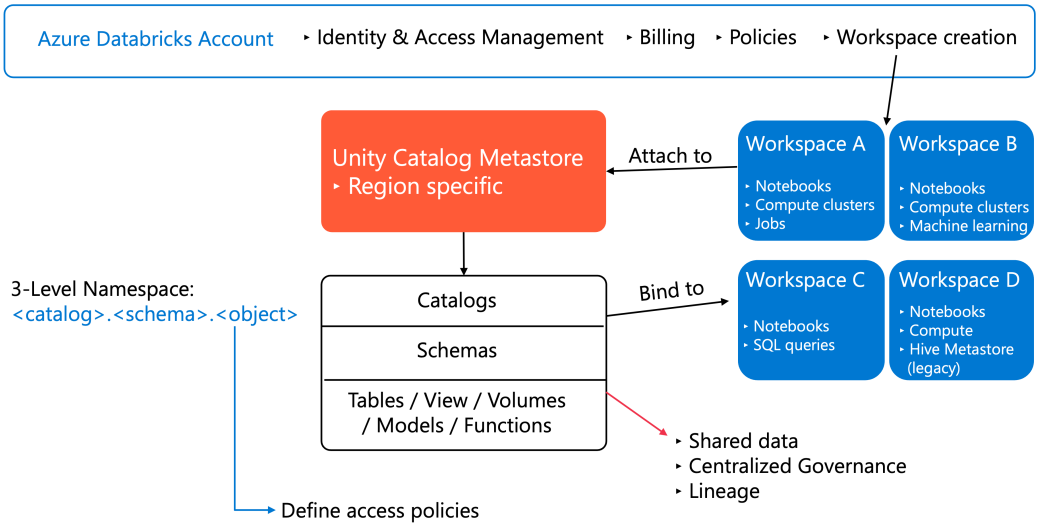
Azure Databricks organizes resources in a hierarchical structure that starts at the account level. The **account** is the top-level construct you use to manage Azure Databricks across your organization. At this level, you manage identity and access, create and configure workspaces, attach Unity Catalog metastores, and oversee billing and policies.

The **metastore** is the top-level container for metadata in Unity Catalog, registering information about your data and AI assets such as tables, views, volumes, models, and functions, along with the permissions that govern access to them. Each metastore is region-specific, and workspaces attached to the same metastore share a unified view of data, enabling centralized governance across your organization.

Unity Catalog organizes data using a three-level namespace structure: `<catalog-name>.<schema-name>.<object-name>`. Unlike the legacy Hive metastore that operates per workspace, Unity Catalog metastores operate at the account level, allowing you to define data access policies once and apply them consistently across all attached workspaces.

Before Unity Catalog, Azure Databricks used the **Hive metastore (legacy)** as a workspace-level metadata store. When a workspace is enabled for Unity Catalog, the legacy Hive metastore becomes accessible as a catalog named `hive_metastore`, allowing you to query existing data using the format `hive_metastore.<schema-name>.<table-name>`. Databricks recommends migrating to Unity Catalog for enhanced security, centralized governance, and built-in auditing.

You create multiple **workspaces** within an account. Workspaces are the collaboration environments where you and your team run compute workloads such as data ingestion, interactive exploration, scheduled jobs, and machine learning training. Each workspace provides an isolated environment for your projects while sharing the same account-level governance.



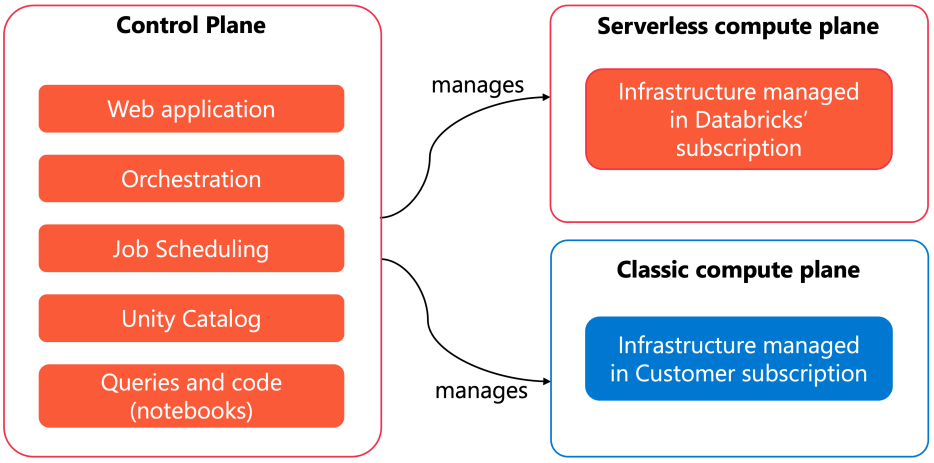
This hierarchical approach allows you to separate concerns: manage organization-wide policies at the account level, isolate projects in workspaces, and govern data centrally through Unity Catalog.

Control plane and compute plane

Azure Databricks separates its architecture into two distinct planes: the control plane and the compute plane.

The **control plane** includes the backend services that Azure Databricks manages in your Azure Databricks account. This plane hosts the web application you use to interact with Azure Databricks, manage configurations, and monitor your workloads. The control plane handles orchestration, job scheduling, and cluster management, but it doesn't process your data.

The **compute plane** is where your data processing happens. Azure Databricks offers two types of compute planes, each designed for different use cases.



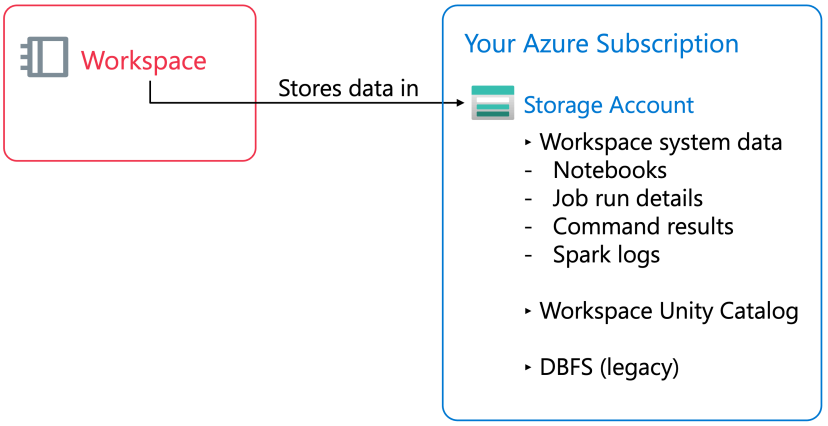
With **serverless compute**, your compute resources run in a serverless compute plane within your Azure Databricks account, not in your Azure subscription. Azure Databricks fully manages the infrastructure, automatically scaling resources based on your workload demands. The serverless compute plane includes network boundaries to isolate workspaces and additional security controls between clusters. This option simplifies operations because you don't need to manage virtual networks or compute resources yourself.

With **classic compute**, your compute resources run in your own Azure subscription in what's called the classic compute plane. Azure Databricks creates new compute resources within each workspace's virtual network in your subscription. This approach provides natural isolation because resources run in your subscription, giving you more control over networking and security configurations.

The separation between control and compute planes allows Azure Databricks to manage orchestration centrally while processing your data in isolated, secure environments.

Workspace storage

Each Azure Databricks workspace has an associated **workspace storage account** that resides in your Azure subscription. This storage account serves multiple purposes and contains different types of data.



The workspace storage account contains **workspace system data** generated as you use Azure Databricks features. This includes notebook revisions, job run details, command results, and Spark logs. The system uses this data to provide versioning, auditing, and troubleshooting capabilities.

If your workspace was enabled for Unity Catalog automatically, the workspace storage account also contains the default **workspace catalog**. All users in your workspace can create data assets in the default schema within this catalog, providing a convenient starting point for organizing data. Users access this data through Unity Catalog's governance layer and don't have direct access to the underlying storage, ensuring security and proper access control.

The workspace storage account may also contain **DBFS (Databricks File System)**, which is a distributed file system accessible under the `dbfs:/` namespace. DBFS root and DBFS mounts are legacy features. Storing and accessing data using DBFS root or DBFS mounts is a deprecated pattern. Instead, you should use Unity Catalog-managed tables and volumes for better governance and security.

Understanding where your data resides helps you implement appropriate security controls, such as enabling firewall support for your workspace storage account to limit access to authorized resources and networks only.

3. Understand Unity Catalog managed storage

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-architecture/3-understand-unity-catalog-managed-storage

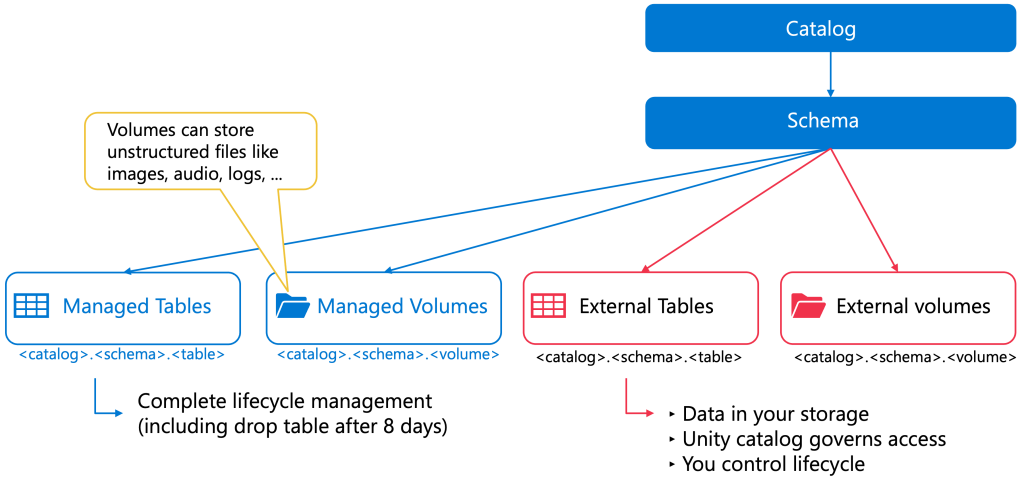
Understand Unity Catalog managed storage

Completed

Your organization likely has data governance policies that require specific types of data to reside in designated cloud storage locations. For example, you might need production data in one storage account and development data in another, or you might need to isolate sensitive customer data from operational data. **Unity Catalog managed storage** helps you meet these requirements while maintaining centralized governance and access control.

What is managed storage?

Managed storage in Unity Catalog refers to cloud storage locations where Unity Catalog stores data and metadata files for **managed tables** and **managed volumes**. Tables store structured data in a tabular format, while **volumes** provide governance for nontabular data files such as images, audio files, logs, or any other unstructured data that doesn't fit into a table schema.



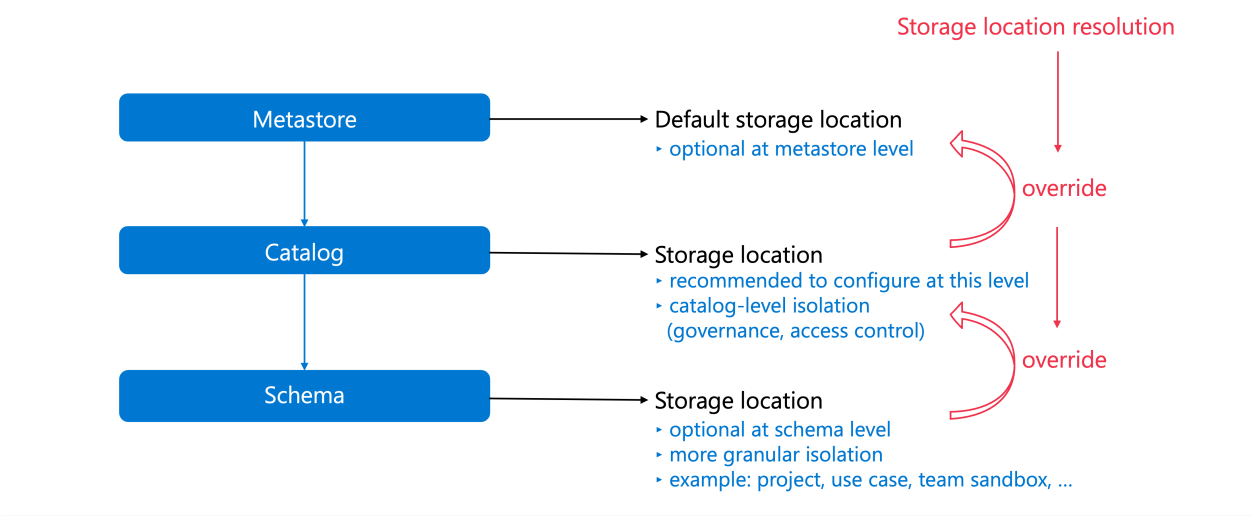
When you create a managed table or volume, Unity Catalog handles the complete lifecycle of the data—including where it's stored, how it's organized, and when it's deleted. This differs from **external tables** and volumes, where you manage the data lifecycle in your cloud storage, and Unity Catalog only governs access from Azure Databricks.

With managed storage, you specify a cloud storage path, and Unity Catalog takes care of the rest. This simplifies data management because you don't need to worry about file layouts or cleanup operations. When you drop a managed table, Unity Catalog automatically marks the underlying data files for deletion after eight days, along with removing the metadata.

Managed storage locations serve two key purposes. First, they provide **physical data isolation** by storing data in separate cloud storage containers or paths. Second, they align with your organizational structure and compliance requirements by letting you map catalogs and schemas to specific storage locations that meet regulatory or security policies.

How managed storage relates to Unity Catalog hierarchy

Unity Catalog organizes data using a **three-level namespace**: **catalog**, **schema**, and **table**. Managed storage locations can be defined at three corresponding levels in this hierarchy: the **metastore level**, the **catalog level**, and the **schema level**. Each level represents a different scope of data organization and isolation.



At the **metastore level**, you can optionally define a default storage location that serves as a fallback for any catalog or schema that doesn't have its own managed storage location. However, new workspaces enabled for Unity Catalog don't automatically include metastore-level storage, and Databricks recommends using **catalog-level storage** instead for better data isolation.

At the **catalog level**, you define storage locations that align with your organizational structure. Catalogs typically represent major organizational units, development lifecycle stages, or data classification categories. For example, you might create a catalog for production data with managed storage in a specific container, and another catalog for development data with storage in a different container. This catalog-level isolation is the recommended approach because it provides a clear boundary for data governance and access control.

At the **schema level**, you can define even more granular storage locations within a catalog. Schemas organize data into logical categories that are more specific than catalogs—typically representing individual projects, use cases, or team sandboxes. By assigning managed storage at the schema level, you can achieve **fine-grained data isolation** when needed.

Storage location resolution

When you create a managed table or volume, Unity Catalog determines where to store the data by following a **resolution hierarchy**. This hierarchy evaluates storage locations from the most specific (schema) to the most general (metastore).

Unity Catalog first checks whether the containing **schema** has a managed storage location. If a schema-level location exists, the data is stored there. This provides the most granular level of control over where your data resides.

If the schema doesn't have a managed storage location, Unity Catalog checks the containing **catalog**. If a catalog-level location exists, the data is stored in that catalog's managed storage location. This is the most common scenario and the recommended approach for most organizations.

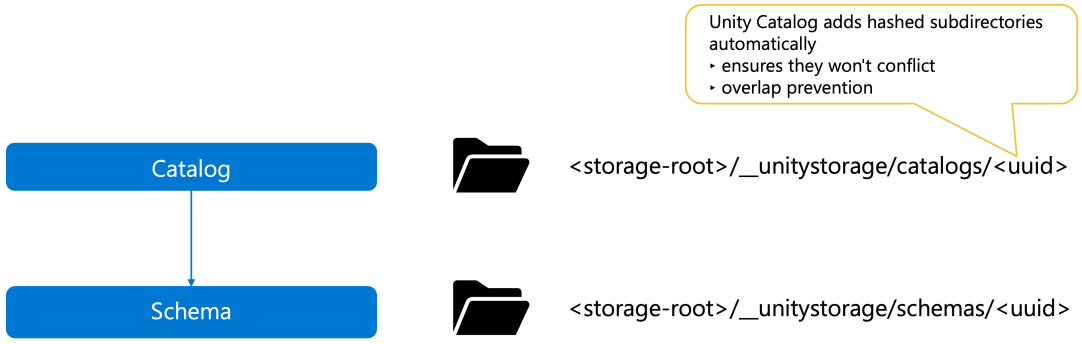
If neither the schema nor the catalog has a managed storage location, Unity Catalog falls back to the **metastore-level** storage location. If no metastore-level location exists either, you won't be able to

create managed tables or volumes, and you'll need to configure a managed storage location at one of these levels first.

This hierarchy gives you flexibility in how you organize and isolate data. You can start with catalog-level storage for most use cases and add schema-level storage when you need more granular control.

Storage root and storage location

When you define managed storage locations for catalogs or schemas, you specify a cloud storage path that Unity Catalog uses as the **storage root**. However, Unity Catalog doesn't store data directly in this path. Instead, it **automatically adds hashed subdirectories** to ensure each catalog and schema has a unique location, even if multiple objects share the same storage root.



For catalogs, Unity Catalog appends subdirectories in the format `__unitystorage/catalogs/<uuid>`, where `<uuid>` is a unique identifier. For schemas, it uses `__unitystorage/schemas/<uuid>`. The complete path—the storage root plus these automatically generated subdirectories—becomes the **storage location** where managed tables and volumes actually store their data.

This automatic path management provides important benefits. You can configure multiple catalogs or schemas with the same base storage root, and Unity Catalog ensures they won't conflict by creating distinct subdirectories for each. You don't need to manually organize or manage these paths—Unity Catalog handles this automatically while maintaining the data isolation and governance you need.

Unity Catalog also enforces **overlap prevention rules** to maintain data governance integrity. When you create a catalog or schema with a managed storage location, Unity Catalog validates that the storage path doesn't overlap with other managed storage locations, external tables, or external volumes. This prevents conflicts and ensures clear boundaries between different data assets. If you attempt to create a managed storage location that overlaps with an existing location, Unity Catalog will reject the operation to protect your data governance model.

4. Understand external storage

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-architecture/4-understand-external-storage

Understand external storage

Completed

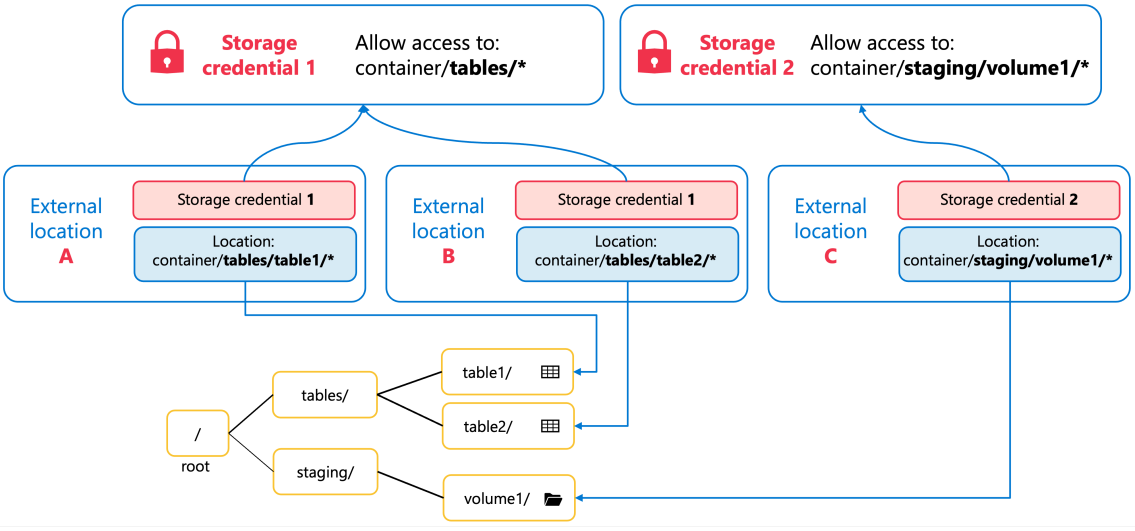
As a data engineer, you need to connect Azure Databricks to data stored in cloud storage locations like Azure Data Lake Storage containers. You might have production data in one storage account and development data in another, or you might need to work with data that's managed by teams outside of Databricks. **External locations** in Unity Catalog allow you to securely connect cloud storage to your workspaces while maintaining governance and access control.

What are external locations?

External locations are Unity Catalog objects that define secure connections to cloud storage. Each external location combines two components: a cloud storage path and a storage credential that authorizes access to that path.

A **storage credential** represents an authentication mechanism, such as an Azure managed identity or service principal. The storage credential provides the authentication required to access your cloud storage. Storage credentials are defined once in Unity Catalog and reference the identity configured in your Azure tenant.

An external location specifies which storage credential to use and which **cloud storage path** to access. This separation between credentials and locations allows one storage credential to be referenced by multiple external locations if they all access storage in the same security boundary.



External locations can reference storage in Azure Data Lake Storage containers, AWS S3 buckets (read-only), or Cloudflare R2 buckets. For Azure Databricks workloads, you'll typically use Azure Data Lake Storage containers because they integrate natively with Unity Catalog and support both read and write operations.

Why do we need external locations?

External locations serve two purposes in Unity Catalog: they enable you to work with external data assets, and they allow you to define managed storage locations at different levels of your data hierarchy.

External tables and volumes let you work with data that's stored and managed outside of Unity Catalog. When you create an external table, you're essentially registering existing data in cloud storage so that Unity Catalog can govern access to it. The data files remain in their original location, managed by your cloud provider or other data platforms. This approach is useful when you have large amounts of existing data, when other systems also need to access the data, or when you want to maintain control over the data lifecycle outside of Databricks.

Managed storage locations use external locations to define where Unity Catalog should store managed tables and volumes. Even though the data is managed by Unity Catalog, it still needs to reside in cloud storage that you own. An external location specifies the storage path, which is then assigned as the managed storage location for a catalog or schema. This allows you to determine where your data physically resides while Unity Catalog handles the data lifecycle.

The key difference lies in who manages the data. With external tables and volumes, you manage the data files and Unity Catalog governs access. With managed storage locations, Unity Catalog manages both the data lifecycle and access control, but stores the data in a location you specify through an external location.

Aspect	External Tables/Volumes	Managed Storage Locations
Data Management	You manage the data files	Unity Catalog manages the data
Data Location	Data remains in original location	Data stored in location you specify
Use Case	Existing or shared data	New data managed by Unity Catalog

Understand storage credentials

Before you can create external locations, you need **storage credentials**—Unity Catalog securable objects that encapsulate the authentication mechanism for accessing cloud storage. Storage credentials centralize credential management. Instead of handling authentication in notebooks or queries, you define credentials once in Unity Catalog and reference them across multiple external locations.

Azure managed identities are the recommended authentication mechanism. A managed identity is an Azure resource that provides an identity for applications connecting to resources that support

Microsoft Entra ID authentication. Azure automatically manages the credential lifecycle—no passwords, no secrets to rotate, and support for storage accounts protected by network firewalls.

Service principals are a legacy alternative. They require you to create an application identity in Microsoft Entra ID and manage client secrets manually, including periodic rotation. Managed identities are preferred for these reasons.

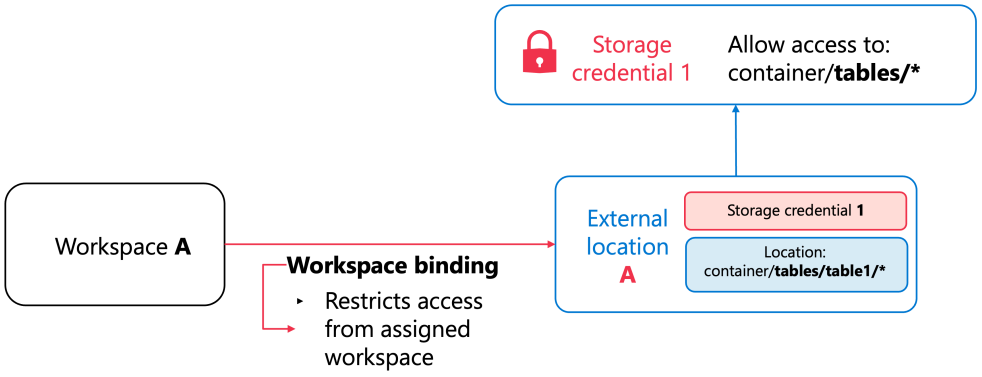
Storage credentials are metastore-level objects available to all attached workspaces. Only users with appropriate privileges can use them to create external locations.

> **Note**
>
> A detailed walkthrough of creating and configuring external storage will be covered in a subsequent module.

Workspace binding for external locations

By default, an external location is accessible from all workspaces attached to your metastore. This means that any user with appropriate privileges can use the external location from any workspace. However, you might want to restrict access to specific workspaces to align with your organization's security boundaries or data governance policies.

Workspace binding restricts an external location to designated workspaces. When enabled, users can only access the external location from assigned workspaces, regardless of their Unity Catalog privileges. This creates an additional layer of access control beyond user-level permissions.



Workspace binding is relevant when workspaces represent different environments (production versus development) or when compliance requirements mandate that certain data remains accessible only from specific compute environments. For example, production data can be bound exclusively to production workspaces, preventing access from development environments even for privileged users.

5. Understand default storage (serverless compute)

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-architecture/5-understand-default-storage

Understand default storage (serverless compute)

Completed

When you deploy Azure Databricks workspaces, you typically need to configure cloud storage accounts, set up access credentials, and manage storage permissions. **Default storage** in Azure Databricks simplifies this process by providing ready-to-use, fully managed storage that's automatically available in serverless workspaces.

What is default storage?

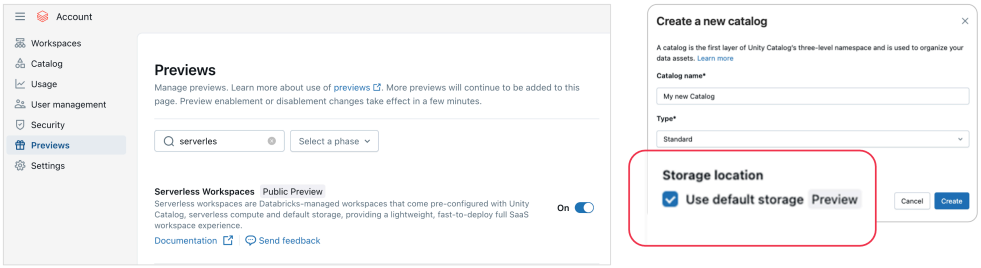
Default storage is a fully managed object storage platform built into Azure Databricks. It provides immediate storage capabilities without requiring you to configure external cloud storage accounts or manage access credentials.

This storage platform serves multiple purposes within your Azure Databricks environment. It stores workspace-level artifacts, control plane metadata, and data for catalogs you create. Unlike external storage where you maintain separate cloud storage accounts, default storage is managed entirely by Azure Databricks.

When you create a serverless workspace, Azure Databricks automatically provisions a default catalog that uses default storage. You can also create additional catalogs that use either default storage or your own cloud object storage, giving you flexibility in how you organize your data.

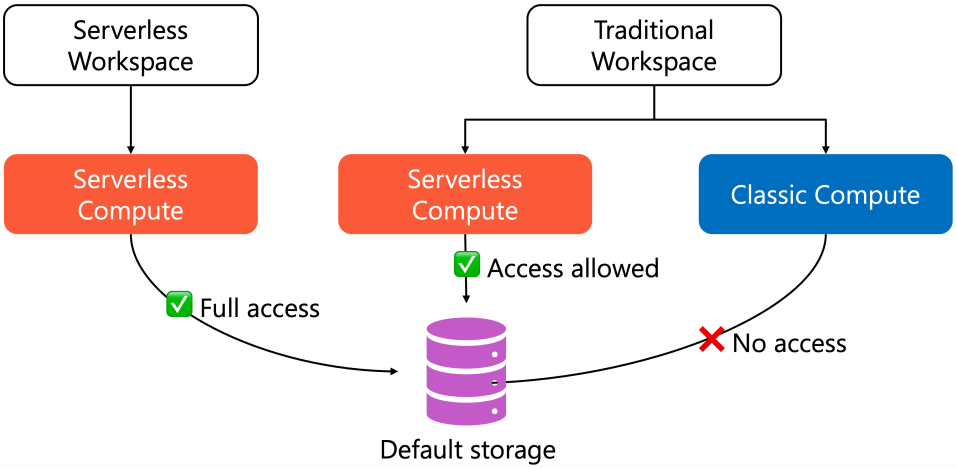
Where is default storage available?

Default storage is available exclusively in serverless workspaces.



Serverless workspaces use default storage for three key areas. First, they use it for internal workspace operations and control plane metadata. Second, they store workspace-level files and artifacts there. Third, catalogs you create can use default storage to store managed tables and volumes.

Traditional workspaces can access catalogs stored in default storage, but only when using serverless compute. This means you can share data across workspace types while maintaining the requirement for serverless compute when accessing default storage.



Default storage benefits and considerations

The following table summarizes the key benefits and considerations when using default storage:

Feature	Benefits	Considerations
Configuration and setup	No need to create separate cloud storage accounts, configure access credentials, or manage storage permissions. Azure Databricks handles infrastructure automatically, enabling immediate data access.	Only available in serverless workspaces. Not suitable for organizations that require full control over storage infrastructure.
Compute requirements	Serverless compute provides instant, scalable resources without provisioning. Consistent compute model across all default storage access.	All access requires serverless compute. Classic compute clusters cannot read from or write to catalogs on default storage. Existing classic compute workloads need migration.
Governance and security	Catalogs integrate with Unity Catalog's privilege model. Manage access using SQL GRANT statements instead of cloud storage RBAC (Role Based Access Control), creating a consistent security model across all data assets.	Access control limited to Unity Catalog mechanisms. Organizations requiring cloud-native RBAC integration should use external storage.
Workspace isolation	Catalogs are only accessible from the workspace where they were created by	Cross-workspace access requires explicit catalog binding configuration. Plan

Feature	Benefits	Considerations
	default. Provides natural isolation for development, testing, and production environments.	workspace architecture carefully to ensure data accessibility where needed.
Data lifecycle management	Automatic cleanup of underlying data files when you drop managed tables or volumes. Prevents orphaned data and reduces storage costs over time.	Only applies to managed tables and volumes. External tables must still manage their own storage lifecycle.
External access	BI tools like Power BI and Tableau can access data through Azure Databricks ODBC and JDBC drivers.	External tools that directly read Delta Lake or Iceberg metadata files cannot access default storage. Direct file access not supported. External data pipelines that read files directly need external storage.

Default storage works best for new serverless workloads, development environments, and features that specifically require it. For production workloads with external access requirements or those using classic compute, external storage provides more flexibility.

6. Module assessment

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-architecture/6-knowledge-check

Module assessment

Completed

7. Summary

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-architecture/7-summary

Summary

Completed

Azure Databricks architecture provides a flexible foundation for data engineering workloads through its hierarchical organization and separation of **control and compute planes**. Throughout this module, you explored how accounts organize resources across workspaces and metastores, how **Unity Catalog** governs data centrally, and how different compute models serve specific needs. The **control plane** handles orchestration while **compute planes**—whether **serverless** or **classic**—process your data in isolated, secure environments.

Storage patterns in Azure Databricks have evolved to address different scenarios. **Default storage** simplifies development by providing fully managed storage in serverless workspaces without configuration overhead, though it requires serverless compute for access. **External locations** bridge Unity Catalog with your existing cloud storage, enabling governance over data managed outside Azure Databricks while maintaining security through storage credentials. **Unity Catalog managed storage** gives you control over data placement at catalog and schema levels while Unity Catalog handles the complete lifecycle, including automatic cleanup of managed tables and volumes.

Understanding these architectural components helps you design Azure Databricks environments that align with organizational requirements. The **account hierarchy** provides clear governance boundaries, the separation of planes enables secure scaling, and the storage options let you balance convenience with control. As you implement Azure Databricks solutions, consider how each component serves your specific use cases, evaluate security implications of different storage patterns, and leverage Unity Catalog's governance capabilities to maintain data quality and access control across your organization.

Understand Azure Databricks Integrations

1. Introduction

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-integrations/1-introduction

Introduction

Completed

Azure Databricks excels at large-scale data processing and machine learning, but modern data solutions require integration with business intelligence tools, application frameworks, and AI services. Understanding how Azure Databricks connects with the **Microsoft ecosystem** helps you design comprehensive solutions that leverage the strengths of each platform.

Microsoft provides multiple integration points with Azure Databricks. You can connect **Microsoft Fabric** for unified analytics, publish data to **Power BI** for interactive reporting, develop locally with **Visual Studio Code** while executing on remote clusters, build applications with **Power Platform**, create AI agents in **Copilot Studio**, implement governance with **Microsoft Purview**, and orchestrate AI workflows with **Microsoft Foundry**. These integrations maintain your governance policies while extending data access across your organization.

The key to successful integration lies in choosing the right pattern for each scenario. **Security models** vary—some pass through individual user credentials to enforce fine-grained permissions, while others use **service principals** for consistent access. Each integration preserves **Unity Catalog** as the central governance layer, but enforcement mechanisms differ based on the consuming service's architecture.

In this module, you explore seven integrations between Azure Databricks and Microsoft services. You learn how each works, when to apply it, and what security considerations matter while maintaining the governance and quality standards your organization requires.

2. Understand integration with Microsoft Fabric

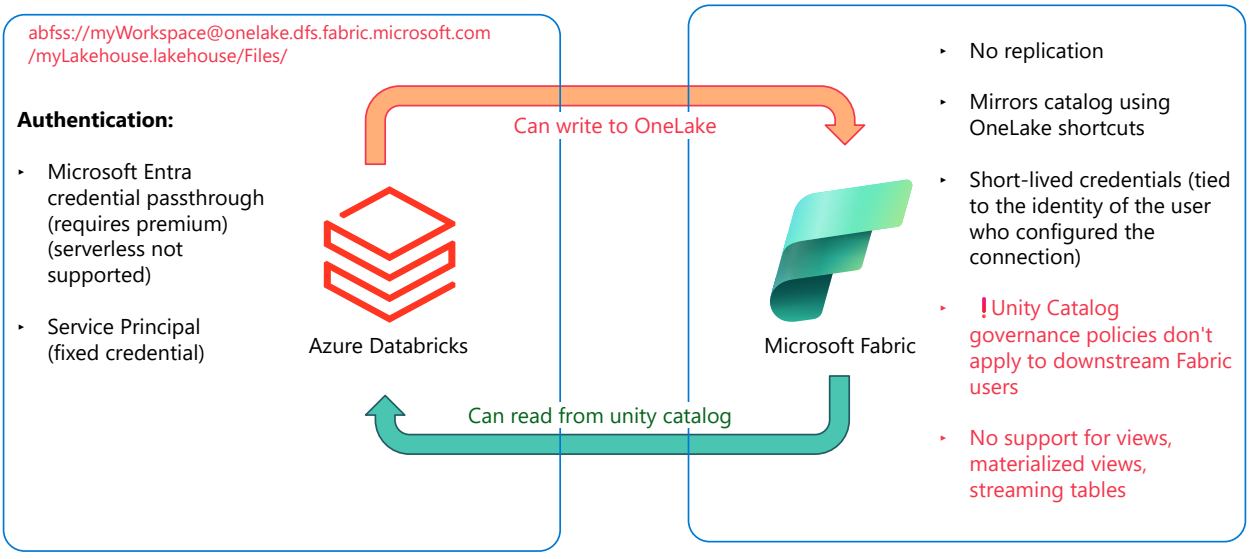
https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-integrations/2-integration-databricks-microsoft-fabric

Understand integration with Microsoft Fabric

Completed

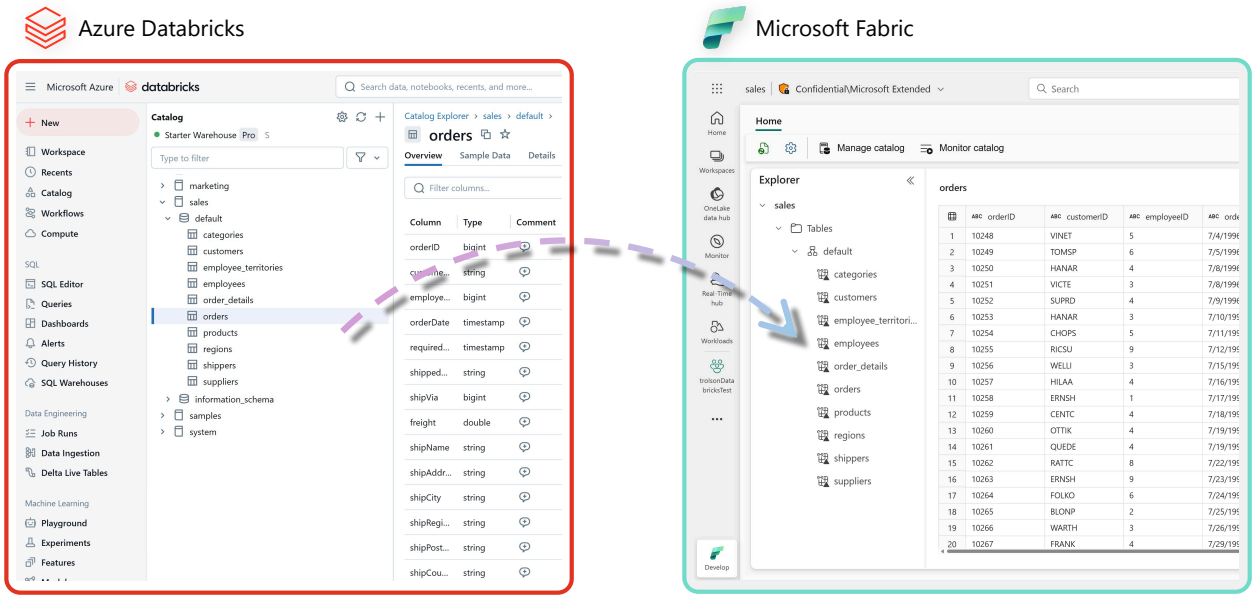
Microsoft Fabric and Azure Databricks complement each other in modern data architectures. Both platforms excel at data engineering and advanced analytics, with Azure Databricks specializing in large-scale data processing and machine learning, while Fabric offers a unified platform with integrated business intelligence and reporting capabilities. Understanding how these platforms integrate helps you design solutions that leverage the strengths of both.

The integration between Azure Databricks and Microsoft Fabric works **bidirectionally**. Fabric can **read** data registered in Unity Catalog, and Azure Databricks can **write** data to OneLake. Each integration pattern serves different scenarios and comes with specific considerations for governance and security.



Access Unity Catalog data from Fabric

Fabric can read tables registered in Unity Catalog **without replicating the data**. When you create a **Mirrored Azure Databricks Catalog** item in Fabric, it mirrors the catalog structure and automatically creates **OneLake shortcuts** for each table. These shortcuts point to the Delta tables stored in Azure Data Lake Storage. The mirroring process doesn't copy your data. Instead, Fabric uses Unity Catalog's open APIs to obtain credentials for accessing the underlying storage paths.



The authentication mechanism relies on **short-lived credentials** that Unity Catalog provides to Fabric. These credentials refresh every hour and can be revoked through Unity Catalog at any time. This approach maintains data in a **single location** while allowing Fabric users to query it.

However, this integration has an **important security implication**. Fabric engines perform authorization using the identity of the user who configured the connection, **not the identity of users who query the data**. **Unity Catalog governance policies don't apply to downstream Fabric users**. Once a table is exposed in Fabric, any Fabric user with access to the connection can query it, regardless of Unity Catalog permissions. Evaluate this against your organization's security requirements.

The user who configures the Fabric connection must have the `EXTERNAL USE SCHEMA` **privilege** on the schemas in Unity Catalog that contain the tables to share. This privilege allows Unity Catalog to issue credentials for the underlying storage. After the connection is established, downstream Fabric users don't need this privilege.

Several **limitations** apply to this integration pattern. Fabric can't access views, materialized views, streaming tables, Delta Sharing catalogs, or tables with row-level filters or column masks. **Only Delta format tables are supported**. The integration doesn't work with workspaces that use private endpoints or IP access lists, and Unity Catalog lineage doesn't track operations performed in Fabric.

Write data to OneLake from Databricks

Azure Databricks can write data directly to Microsoft Fabric lakehouses in OneLake. This integration pattern supports scenarios where you process and transform data in Databricks, then make it available in Fabric for analytics and reporting.

To connect Azure Databricks to OneLake, you use the **Azure Blob Filesystem (ABFS) driver** with OneLake endpoints. The connection path follows the format: `abfss://myWorkspace@onelake.dfs.fabric.microsoft.com/myLakehouse.lakehouse/Files/`.

This path structure identifies your Fabric workspace and lakehouse, allowing Databricks to read from and write to specific locations.

You can authenticate to OneLake through **two methods**. With **Microsoft Entra credential passthrough**, user identities flow through to OneLake, but this requires a **premium Azure Databricks workspace** and proper cluster configuration with **ADLS credential passthrough enabled**. Alternatively, you can authenticate using a **service principal**, which uses a fixed credential rather than individual user identities. Service principals work with both traditional clusters and Azure Databricks serverless compute, providing more flexibility for automated workflows and job execution.

> **Note**
>
> Credential passthrough is not supported with serverless compute

The workflow is straightforward. You load data in Databricks, apply transformations using Spark, and write the results to OneLake using standard Spark write operations. Once data lands in your Fabric lakehouse, it becomes immediately available to Power BI, data warehouses, and other Fabric services without additional data movement.

This integration supports **serverless compute**, allowing you to run workloads without provisioning clusters. When using serverless compute, you must use service principal authentication and ensure your code doesn't modify unsupported Spark configuration properties. The service principal needs appropriate workspace role assignments in Fabric to write data.

Integration scenarios

These integration patterns enable several practical scenarios for organizations using both platforms. You can build **data pipelines** where Databricks handles complex transformations and machine learning workloads, then publishes curated datasets to Fabric for business analytics. Data engineers work in Azure Databricks while business analysts access the results through familiar Fabric tools.

With Fabric reading from Unity Catalog, you can make specific datasets available for reporting **without duplicating storage**. Business users create Power BI reports that query data managed in Unity Catalog, maintaining a **single source of truth**. This approach applies when you have data governance established in Unity Catalog and want to extend access to Fabric users for specific tables.

For scenarios requiring **real-time or near-real-time** analytics, Databricks streaming jobs can write processed data to OneLake continuously. Fabric workloads can then consume this data as it arrives. This pattern separates the responsibilities of data engineering and business intelligence while keeping them connected through shared storage.

When you need to combine data from multiple sources, you can use Azure Databricks to **join and transform** data from various systems, then land the integrated results in OneLake. Fabric serves as the presentation layer, providing self-service analytics on the prepared datasets. This architecture lets data engineers focus on data quality and transformation logic while business users create their own analyses.

Key considerations

Security and governance are critical factors when you implement these integrations. The Fabric-to-Unity Catalog integration **bypasses Unity Catalog's fine-grained access controls** for downstream users. Carefully consider which tables to expose and ensure your organization accepts this security model. For scenarios requiring strict governance, consider using **Power BI Direct Query** instead, which honors Unity Catalog permissions.

Performance and cost also matter. Reading Unity Catalog data from Fabric requires a running Fabric capacity for metadata scans and refreshes, which introduces additional costs. When writing from Azure Databricks to OneLake, network transfer costs apply if data crosses region boundaries. Plan your architecture to minimize unnecessary data movement and optimize for your workload patterns.

Storage architecture remains important. Both integration patterns leverage **Azure Data Lake Storage** as the underlying storage layer. Azure Databricks uses Unity Catalog to manage and govern this storage, while Fabric accesses it through OneLake abstractions. Understanding this **shared foundation** helps you design efficient solutions that avoid data duplication and maintain consistency.

The choice between integration patterns depends on your specific requirements. Use Fabric reading from Unity Catalog when you want to expose existing Databricks-managed data to Fabric users and can accept the security model. Use Azure Databricks writing to OneLake when you need to publish curated datasets specifically for Fabric consumption and want more control over what data is shared. In many architectures, you might use both patterns for different scenarios within the same organization.

3. Understand integration with Power BI

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-integrations/3-integration-databricks-power-bi

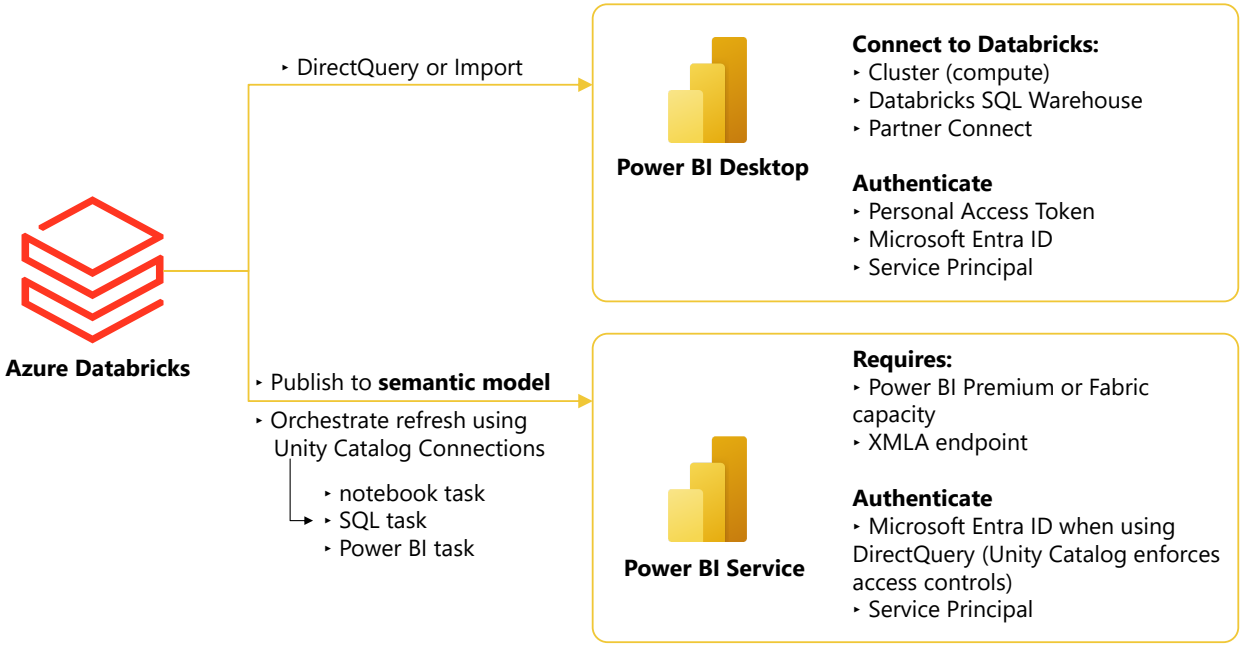
Understand integration with Power BI

Completed

Power BI provides interactive visualizations and business intelligence capabilities. Integrating Power BI with Azure Databricks allows users to create reports and dashboards from Databricks-managed data. This integration supports multiple user roles, from data engineers to business analysts.

The integration between Azure Databricks and Power BI works through **multiple connection methods**. You can connect Power BI Desktop to your compute resources, publish data directly from Azure Databricks to the Power BI service, or orchestrate semantic model refreshes using Unity

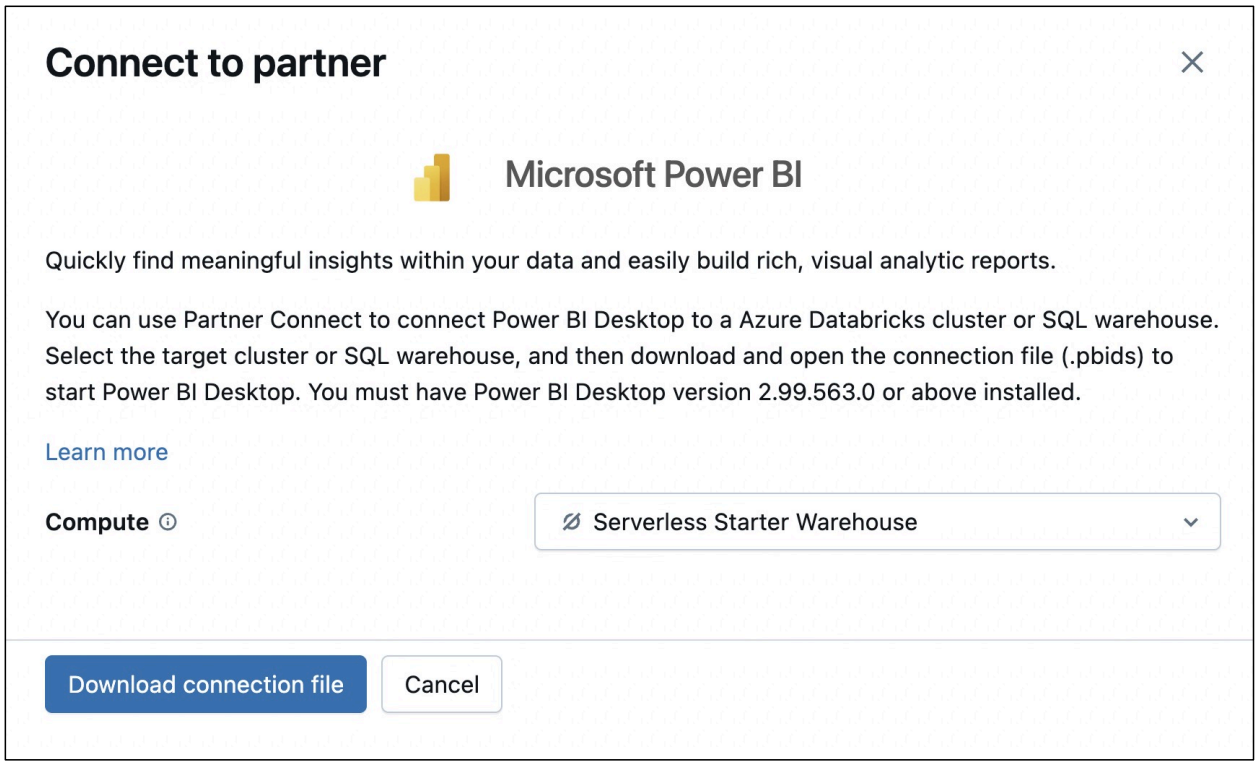
Catalog connections. Each method serves different scenarios and supports various authentication options.



Connect Power BI Desktop to Azure Databricks

Power BI Desktop is a Windows-based application that enables you to create interactive reports and dashboards. You can connect Power BI Desktop to both Azure Databricks **clusters** and **Databricks SQL warehouses**. **Use SQL warehouses with DirectQuery mode** for optimized query performance and built-in serverless scaling.

The fastest way to establish a connection is through **Partner Connect**. From the Azure Databricks workspace Marketplace, you select the Power BI tile, choose your compute resource, and download a connection file. This file opens Power BI Desktop with preconfigured connection settings. Alternatively, you can configure the connection manually by entering the **Server Hostname** and **HTTP Path** from your compute resource details.



After establishing the connection, you select your **data connectivity mode**. With **Import** mode, Power BI loads data into its internal storage engine, enabling fast queries but requiring periodic refreshes to stay current. With **DirectQuery** mode, Power BI queries Azure Databricks directly for each visualization, ensuring real-time data access but requiring an active connection and running compute resource.

Authentication determines how Power BI accesses your data. **Personal access tokens** provide simple authentication but represent a single user identity. **Microsoft Entra ID** enables single sign-on where each user's credentials pass through to Azure Databricks, allowing Unity Catalog to enforce fine-grained permissions. For automated scenarios, **service principals** with machine-to-machine OAuth provide fixed credentials that don't depend on individual user accounts.

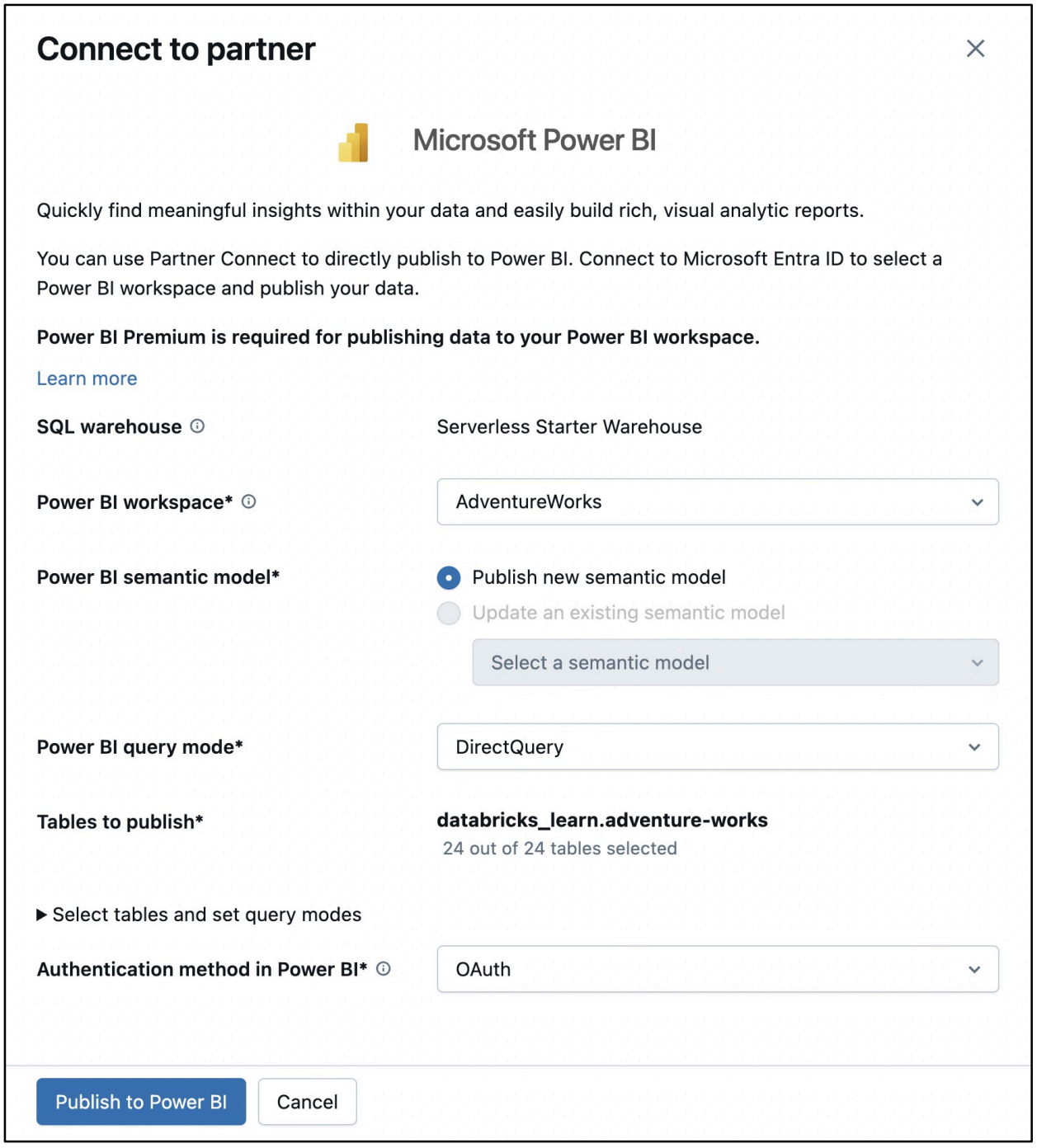
Power BI Desktop also supports **native SQL queries** for compute-intensive operations. Instead of using Power BI's visual query builder, you write SQL directly against your Databricks SQL warehouse. This approach gives you full control over query optimization and access to Databricks-specific SQL features.

> **Note**
>
> Power BI Desktop requires Windows. If you use a different operating system, run Power BI Desktop on a Windows-based virtual machine and connect to it remotely.

Publish to the Power BI service from Azure Databricks

The Power BI service is a cloud-based platform where you share reports and collaborate with your organization. Rather than connecting Power BI to Databricks, you can publish data **from** Azure Databricks directly to the Power BI service. This workflow simplifies the publishing process by initiating it from the Azure Databricks UI.

To publish data, you use **Catalog Explorer** in your Azure Databricks workspace. You select a schema or specific tables from Unity Catalog, then choose **Publish to Power BI workspace**. The publishing wizard authenticates you with Microsoft Entra ID and prompts you to select your target Power BI workspace, choose between DirectQuery and Import modes, and configure authentication settings.



This publishing method requires **Unity Catalog** for data governance and a **Power BI Premium license** (Premium capacity, Premium Per User, or Microsoft Fabric capacity). The Premium license

provides the necessary capacity for enterprise scenarios and enables the XMLA (XML for Analysis) endpoint that Databricks uses to create and update semantic models programmatically.

When you publish a schema containing multiple tables, Databricks creates a **semantic model** in your Power BI workspace. **Column comments** from Unity Catalog tables transfer to Power BI as column descriptions, preserving your documentation. **Foreign key relationships** also transfer, though Power BI supports only one active relationship path between any two tables, so some relationships might become inactive if multiple paths exist.

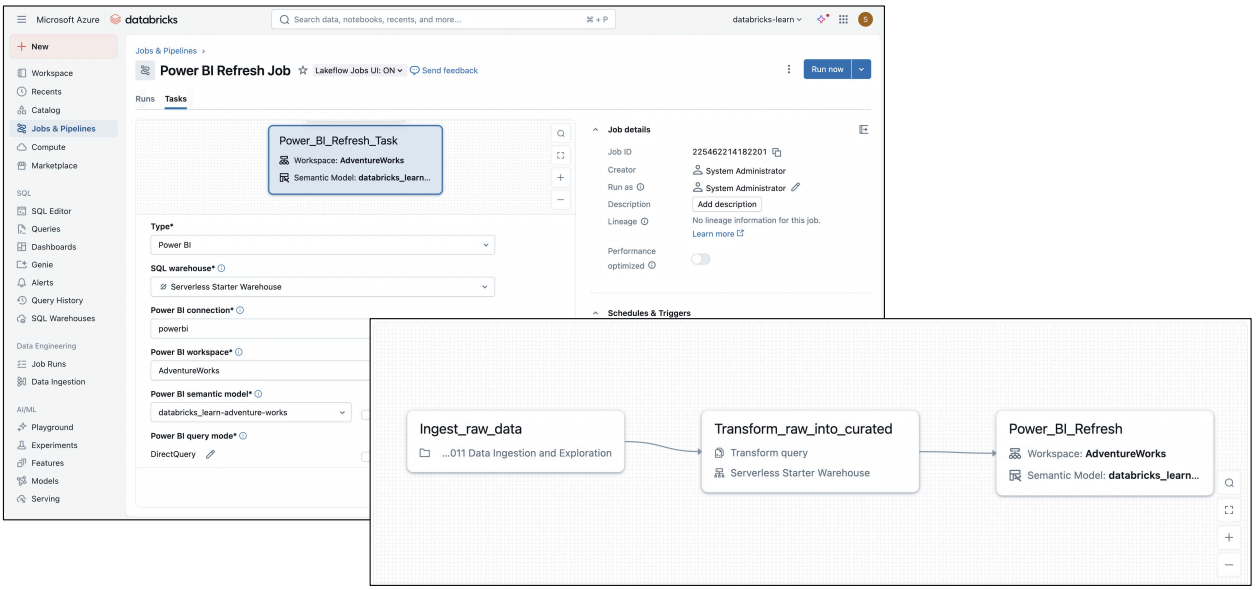
Use OAuth authentication for published semantic models to enable fine-grained access control and user-level auditing. When users access a report built on DirectQuery mode, Power BI passes their Microsoft Entra ID credentials to Azure Databricks. Unity Catalog evaluates permissions for that specific user, enforcing row-level filters, column masks, and table access controls. This approach maintains governance consistency across platforms.

For scenarios requiring automated refreshes or shared credentials, you can configure **machine-to-machine OAuth** using service principals. After publishing, you edit the semantic model's data source credentials in Power BI and provide the service principal's client ID and secret. This method works with both semantic models and Power BI gateways for private network scenarios.

> **Important**
>
> Publishing to Power BI service requires the XMLA endpoint to be enabled with Read and Write capabilities in your Power BI/Fabric capacity settings.

Automate Power BI publishing with Power BI tasks

While you can publish to Power BI manually from Azure Databricks, **Power BI tasks** enable you to orchestrate semantic model publishing and refreshes automatically as part of your Databricks workflows. This automation integrates Power BI updates into your data pipelines, ensuring that reports refresh only after upstream data transformations complete.

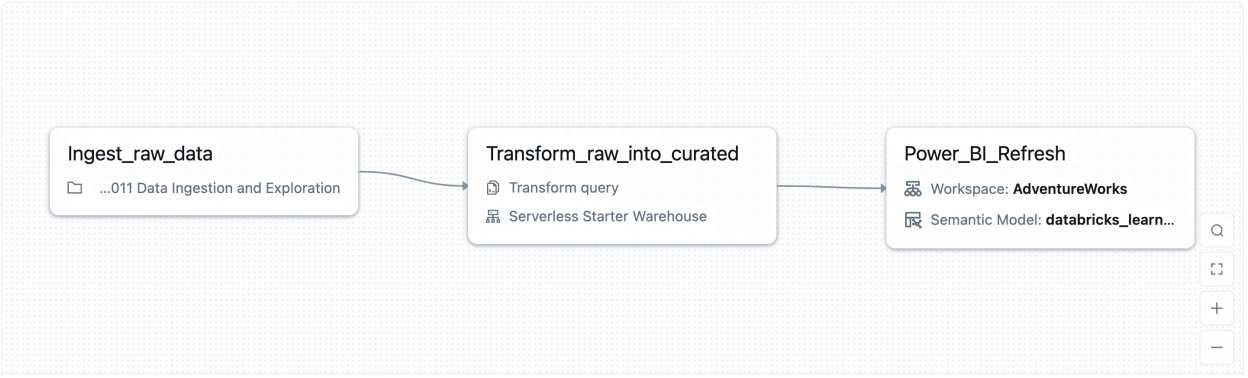


Power BI tasks require a **Unity Catalog connection** to Power BI. This connection stores authentication credentials and workspace information that the task uses to publish or update semantic models. You choose an authentication method (service credentials, OAuth machine-to-machine, or OAuth user-to-machine) based on your security requirements.

Power BI tasks integrate into Databricks workflows alongside data engineering tasks, machine learning model training, and other automation. You create task dependencies to control execution order, ensuring that Power BI semantic models refresh only after source data completes processing.

For example, a workflow might include:

- A **notebook task** that ingests raw data from external sources into Unity Catalog tables
- A **SQL task** that transforms the raw data into curated analytics tables
- A **Power BI task** that publishes the curated tables to a semantic model and triggers a refresh



When upstream data updates complete successfully, the Power BI semantic model automatically refreshes with the latest data.

Choose the right integration method

The choice between integration methods depends on your workflow and requirements. **Use Power BI Desktop connections** when business analysts need to explore data interactively and build reports. Desktop provides a report authoring environment with drag-and-drop visualizations and the flexibility to combine multiple data sources.

Use publishing from Azure Databricks to Power BI service when you want to share curated datasets with specific governance and want to initiate sharing from the data engineering side. This method ensures that data engineers control which tables become available in Power BI and can document them with column descriptions that transfer automatically.

Orchestrate Power BI updates with data pipelines when you need to automate semantic model refreshes as part of larger data pipelines. This approach coordinates Power BI updates with upstream data transformations, ensuring that reports refresh only after source data completes processing.

For organizations with private network requirements, Power BI gateways enable secure connectivity without exposing Azure Databricks endpoints publicly. You configure an on-premises or virtual network gateway with your Azure Databricks connection details and authentication credentials. Power BI service routes queries through the gateway, which connects to your private Azure Databricks workspace.

Security and governance considerations

Security architecture differs between connection methods and authentication types. The following table compares how Unity Catalog enforces permissions and tracks access:

Use case	Authentication method	Unity Catalog permission enforcement	Audit granularity
Interactive exploration with Power BI Desktop or DirectQuery reports requiring user-specific governance.	**Microsoft Entra ID (SSO)**	Evaluates permissions per user. Row-level filters, column masks, and table access controls apply based on each viewer's identity.	User-level audit logs show which specific users accessed which data.
Scenarios where all users should see identical data, or when implementing row-level security within Power BI.	**Import mode or no SSO**	Uses fixed credentials. All report viewers see the same data regardless of their Unity Catalog permissions.	Limited to the credential owner's identity. Can't track individual viewer access.
Automated workflows, scheduled refreshes, and scenarios where consistent data access is required.	**Service principals**	Evaluates permissions based on the service principal's grants, not individual viewers.	Shows only that the service principal accessed data, not which report viewers consumed it.
Development and testing.	**Personal access tokens**	Represents a single user identity. Permissions apply to that user only. Not	Shows the token owner accessed data,

Use case	Authentication method	Unity Catalog permission enforcement	Audit granularity
		recommended for production automation.	not the actual end user.

4. Understand integration with VS Code

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-integrations/4-integration-databricks-vscode

Understand integration with VS Code

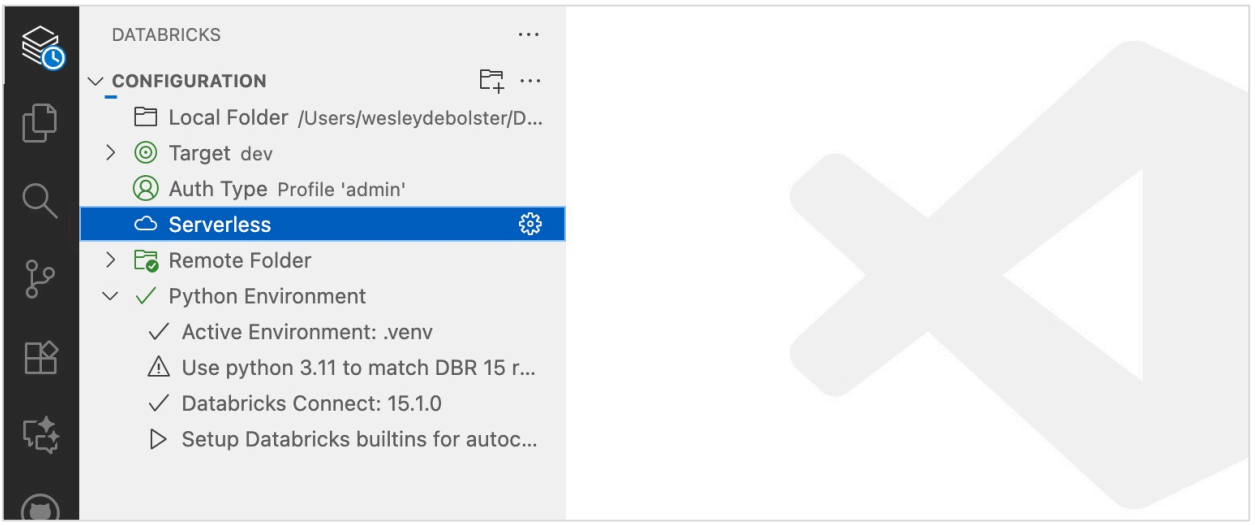
Completed

Data engineers working with Azure Databricks often switch between their local development environment and remote workspaces. You write code locally in your preferred editor, then copy it to Databricks notebooks or upload files to test on clusters. This back-and-forth workflow slows down development and makes debugging difficult. The Databricks extension for Visual Studio Code changes this by connecting your local development environment directly to your remote Azure Databricks workspace.

With this integration, you develop in Visual Studio Code using familiar tools and shortcuts while executing code on Azure Databricks compute resources. You don't need to leave your editor to run notebooks, test Python scripts, or deploy workflows. This approach keeps your development environment consistent while taking advantage of Databricks' distributed computing power.

Develop locally and execute remotely

Visual Studio Code provides a development environment on your local machine with features like **IntelliSense**, **syntax highlighting**, and **Git integration**. The Databricks extension adds the ability to execute your local code on remote Azure Databricks clusters or serverless compute. You write code in `.py` files or notebooks on your computer, then run them directly on Azure Databricks infrastructure without manually copying files or switching between applications.



You can also run Python, R, Scala, and SQL notebooks as **Lakeflow Jobs**. Instead of executing code interactively on a cluster, you package your notebook as a job that runs on a schedule or trigger. This capability bridges local development and production deployment, letting you test job configurations before committing them to source control.

The extension supports multiple Databricks projects within a single Visual Studio Code workspace. If you work on different Databricks environments or modules, you switch between project configurations without closing and reopening folders. Each project maintains its own workspace connection, authentication settings, and cluster selection.

Debug code with Databricks Connect

The Databricks extension integrates **Databricks Connect** to enable full debugging capabilities within Visual Studio Code.

Databricks Connect creates a direct connection between your local Python environment and a remote Azure Databricks cluster. When you debug code, the Python debugger in Visual Studio Code controls execution on the cluster. You set breakpoints, inspect variables, and step through code just as you would with local Python scripts. The difference is that your code runs on Databricks compute with access to distributed data and Spark APIs.

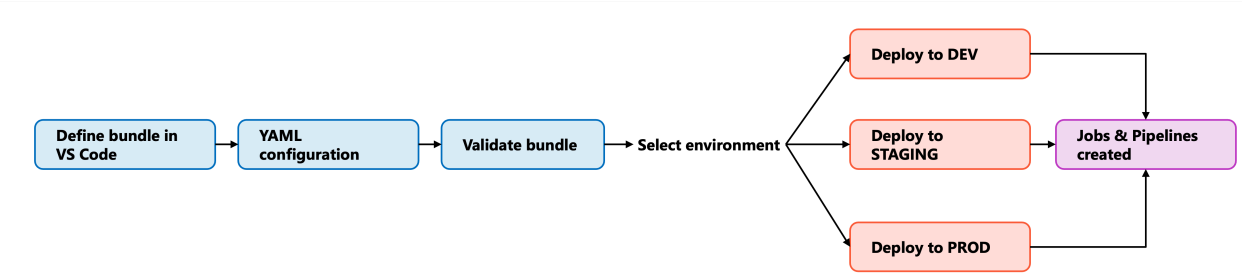
This debugging environment works with both Python files and notebooks. You can debug notebooks cell by cell, examining the state of DataFrames and variables at each step. When you encounter an error, you modify your code locally, set a breakpoint before the problematic line, and rerun the debugger to investigate. This tight feedback loop reduces the time spent troubleshooting data pipelines and transformations.

Beyond interactive debugging, the extension supports running Python tests with `pytest`. You write unit tests for your data processing logic and execute them on Databricks compute. This capability ensures that your tests run against the same environment and dependencies as your production code, catching environment-specific issues early in development.

Deploy workflows with Asset Bundles

The Databricks extension simplifies deployment through **Databricks Asset Bundles**, which package your code, configurations, and dependencies into deployable units.

Asset Bundles define **Lakeflow Jobs**, **Lakeflow Spark Declarative Pipelines (SDP)**, and **MLOps Stacks** using configuration files. You specify compute settings, schedule triggers, and task dependencies in YAML format. The extension provides a UI for creating, validating, and deploying these bundles without leaving Visual Studio Code.



When you're ready to deploy, the extension applies your bundle to a target environment (development, staging, or production). It creates or updates jobs, configures clusters, and sets up permissions according to your specifications. This declarative approach ensures consistent deployments and makes it easier to apply CI/CD patterns to your data engineering workflows.

With bundles, you can version your entire workflow configuration alongside your code in Git. Changes to job schedules, cluster sizes, or task dependencies go through the same review and approval process as code changes. This practice improves collaboration and provides an audit trail for production deployments.

Synchronize code between local and remote workspaces

While Asset Bundles handle deployment, you might also want to keep local code synchronized with workspace folders for quick prototyping or collaboration. The extension supports one-way automatic synchronization from your local Visual Studio Code project to workspace directories. You edit files locally, and changes automatically upload to the workspace. The workspace files are intended to be transient, so you should avoid making changes directly in the workspace, as these changes won't sync back to your local project.

This synchronization capability works well for teams transitioning from workspace-based development to local development. You maintain existing workspace folders while gradually adopting local development practices and version control. By syncing your local files to the workspace, you can test and run them remotely while keeping your local project as the source of truth for version control in Git.

However, for production workflows, Asset Bundles provide better control and consistency. Synchronization suits exploratory work and quick iterations, while bundles suit structured deployment pipelines with environment-specific configurations.

5. Understand integration with Power Platform

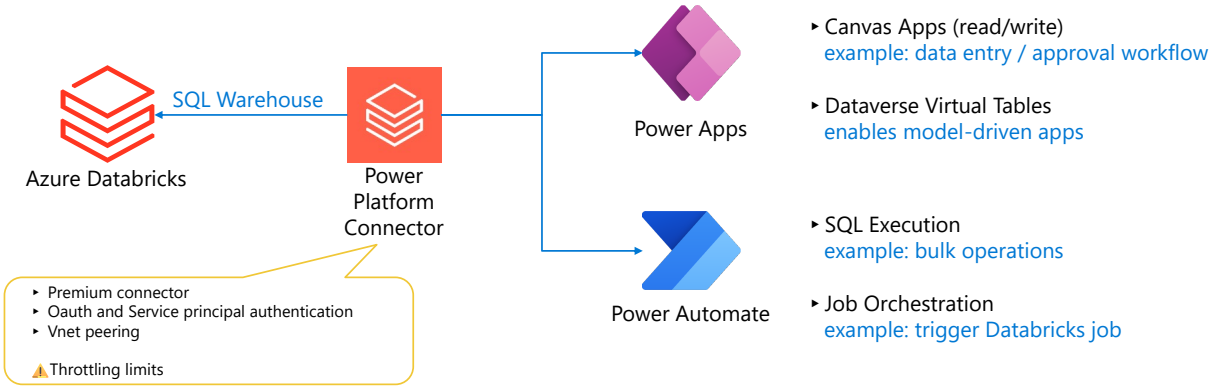
https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-integrations/5-integration-databricks-power-platform

Understand integration with Power Platform

Completed

Power Platform enables you to build custom applications, automate workflows, and analyze data without extensive coding. Integrating Azure Databricks with Power Platform allows you to bring governed, high-quality data from your lakehouse directly into business applications and automation workflows. This integration supports data engineers in delivering data solutions that empower business users across the organization.

The integration between Azure Databricks and Power Platform works through the **Azure Databricks connector**. This connector enables Power Apps and Power Automate to access data stored in Unity Catalog while preserving your governance controls. You can build applications that read and write data, automate job execution, or orchestrate complex data workflows—all while maintaining the security policies and access controls defined in Unity Catalog.



Connect Power Platform to Azure Databricks

Power Platform connects to Azure Databricks through a **premium connector** that requires specific prerequisites. You need a premium Power Apps license, a Microsoft Entra ID account, and access to a SQL warehouse in your Azure Databricks workspace. The connector uses SQL warehouses as the compute resource because they provide optimized query performance and serverless scaling for analytical workloads.

To establish a connection, you create an **Azure Databricks connection** in Power Apps or Power Automate. From the Connections page, you search for the Azure Databricks connector and select

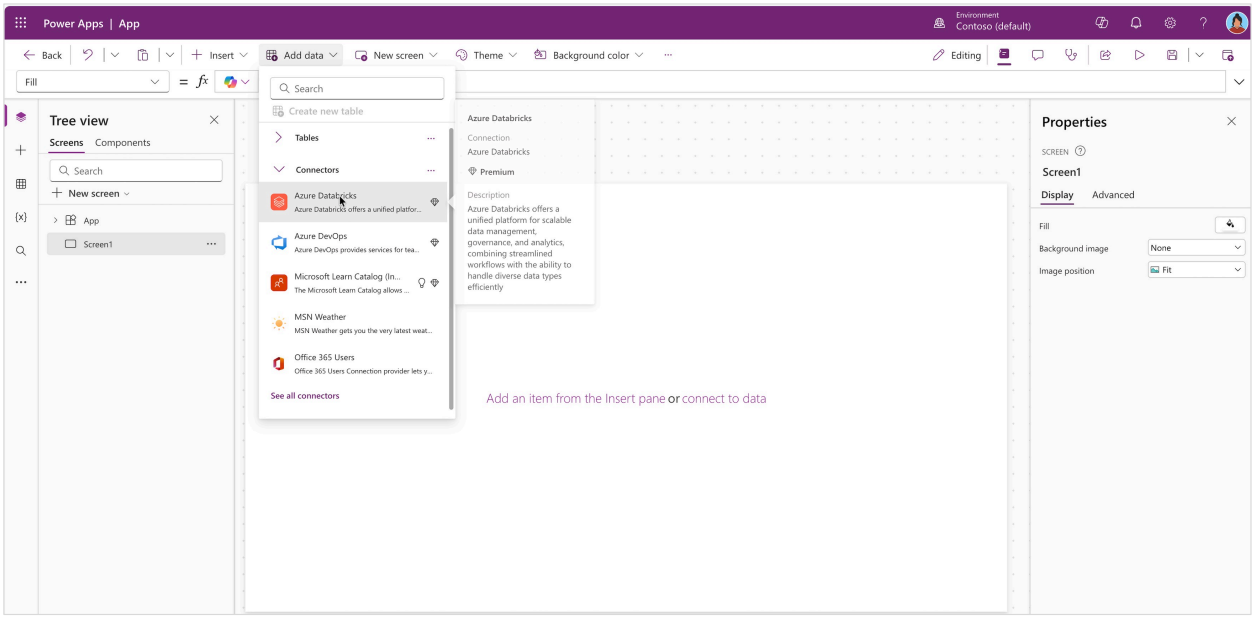
your authentication method. The connector supports two authentication types: **OAuth connection** (which uses Microsoft Entra ID to authenticate individual users) and **service principal connection** (which uses fixed credentials for authentication). Each authentication type serves different scenarios and offers different capabilities.

To complete the connection setup, you provide the **Server Hostname** and **HTTP Path** from your SQL warehouse. These connection details identify which compute resource Power Platform uses to query your data. You can find these values in the connection details section of your SQL warehouse in the Azure Databricks workspace.

> **Note**
>
> If your Azure Databricks workspace uses virtual networks, you can integrate Power Platform using VNet peering for private connectivity or by configuring IP access lists to allow AzureConnectors IP ranges.

Build applications with Power Apps

Power Apps enables you to build **canvas apps** that provide custom interfaces for reading and writing data. When you connect a canvas app to Azure Databricks, you select a catalog and choose the tables that your application will access. The app can perform create, update, and delete operations on your data, making it possible to build data entry forms, approval workflows, or dashboards that interact directly with your lakehouse.



With a direct connection to Azure Databricks, you can build canvas apps that interact with data while **preserving Unity Catalog governance**. When you use OAuth authentication, Unity Catalog evaluates each user's permissions in real time, ensuring that the app respects row-level security, column masking, and other access controls. This approach lets business users work with trusted data through an intuitive interface without compromising security.

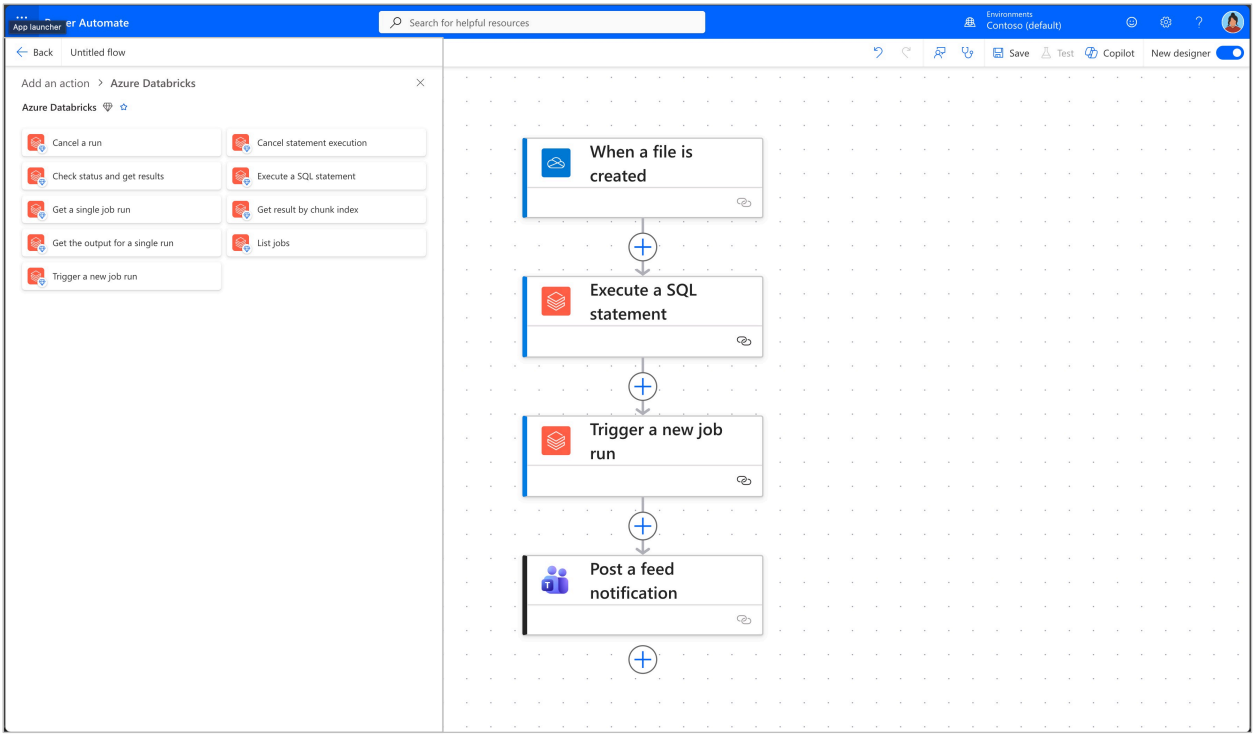
For applications that require **bulk operations**, Power Automate flows can process multiple changes in a batch. This pattern handles large-scale updates more efficiently than individual operations from the canvas app.

Power Apps also supports **Dataverse virtual tables** backed by Azure Databricks data. Virtual tables integrate data from Azure Databricks with Microsoft Dataverse without physically copying the data. This approach enables you to build **model-driven apps** that provide structured forms, views, and business logic on top of your Databricks data.

> **Tip**
>
> For better performance, use direct connections rather than virtual tables when possible. While virtual tables don't consume Dataverse storage capacity, direct connections typically offer faster query response times.

Automate workflows with Power Automate

Power Automate enables you to build **flows** that automate business processes and data operations. The Azure Databricks connector exposes two primary capabilities within Power Automate: **SQL statement execution** for running queries and **job orchestration** for triggering existing Databricks jobs. These capabilities enable you to create automated workflows that respond to business events, schedule data processing tasks, or coordinate complex data pipelines.



With **SQL statement execution**, you can write and execute SQL queries directly from a flow. You create a flow with any trigger type—such as a schedule, a manual trigger, or an event from another

service—then add an Azure Databricks action to execute SQL. The flow can retrieve query results and use them in subsequent steps, with support for handling large result sets through chunked retrieval.

Beyond SQL execution, **job orchestration** lets you trigger existing Azure Databricks jobs from Power Automate. You can start a job run, track its progress, and retrieve metadata about the run including its status, start and end times, and execution duration. This integration enables you to **coordinate data processing workflows** with business processes—for example, triggering a data refresh when new files arrive or sending notifications when a data quality job completes.

The connector also supports canceling running statements or jobs, listing available jobs, and retrieving output from completed runs. These capabilities enable scenarios like automatic retry logic, conditional workflows based on data quality checks, or multi-step data pipelines that combine Databricks processing with other services.

> **Important**
>
> The Power Platform connector has throttling limits. Design your flows to batch operations appropriately when processing large volumes.

Integration scenarios

The Azure Databricks and Power Platform integration enables several practical scenarios that bridge the gap between data engineering and business operations. Consider a **planning and forecasting application** where business users submit next month's forecasts, adjust revenue targets, or update budget allocations through a Power App that writes directly to a Unity Catalog table. With OAuth authentication and Unity Catalog governance, each user sees only the data they're authorized to access, and all changes are tracked in the table's audit log.

Another scenario involves **automated data quality workflows**. You build a Power Automate flow that runs on a schedule, executes a SQL query to check data quality metrics, and sends notifications to the data team when quality thresholds are exceeded. The flow can also trigger a Databricks job to repair data issues automatically or escalate to a human review process based on the severity of the issue.

For **event-driven data processing**, you create a flow that monitors a business system for events—such as a new customer order or a completed transaction—and triggers a Databricks job to process the associated data. The flow passes relevant parameters to the job, waits for completion, and updates the business system with the results. This pattern enables real-time data integration without building custom integration code.

Dashboard refresh automation represents another valuable use case. You schedule a Power Automate flow that executes SQL to prepare aggregated data, stores the results in a table, and triggers a refresh of a business intelligence dashboard. This approach provides business users with up-to-date insights without manual intervention from the data team.

These scenarios become possible because the integration **preserves governance while enabling access**. Unity Catalog continues to enforce all security policies, data masking, and access controls even when data flows through Power Platform. Business users gain the ability to work with data through familiar interfaces while data engineers maintain centralized control over data quality, security, and compliance.

Considerations and limitations

While the Azure Databricks Power Platform connector provides broad capabilities, some considerations affect how you design your solutions. The connector **doesn't support government clouds**, including US Government and China Cloud environments. Organizations operating in these environments need alternative integration approaches.

Within Power Apps, certain **PowerFx formulas** calculate values using only data that has been retrieved locally to the app, rather than delegating computation to Azure Databricks. For large datasets, this can affect performance and accuracy. Consider filtering data or performing aggregations in SQL within Azure Databricks before bringing data into Power Apps.

Concurrent write operations from multiple Power Apps users or flows benefit from row-level concurrency features in Azure Databricks. These features are available in recent Databricks Runtime versions and help reduce write conflicts when multiple users modify data simultaneously.

When choosing between **virtual tables and direct connections**, consider the trade-offs. Virtual tables enable model-driven apps and integrate with the broader Dataverse ecosystem, but they don't support OAuth credential passthrough. Direct connections offer better performance and support OAuth for governance passthrough, but they only work with canvas apps. Choose the approach that best fits your application architecture and governance requirements.

Finally, **data policies** in Power Platform affect connector usage. You can add the Azure Databricks connector to a Business data policy to control which other connectors can share data with it, helping you maintain compliance with data handling regulations.

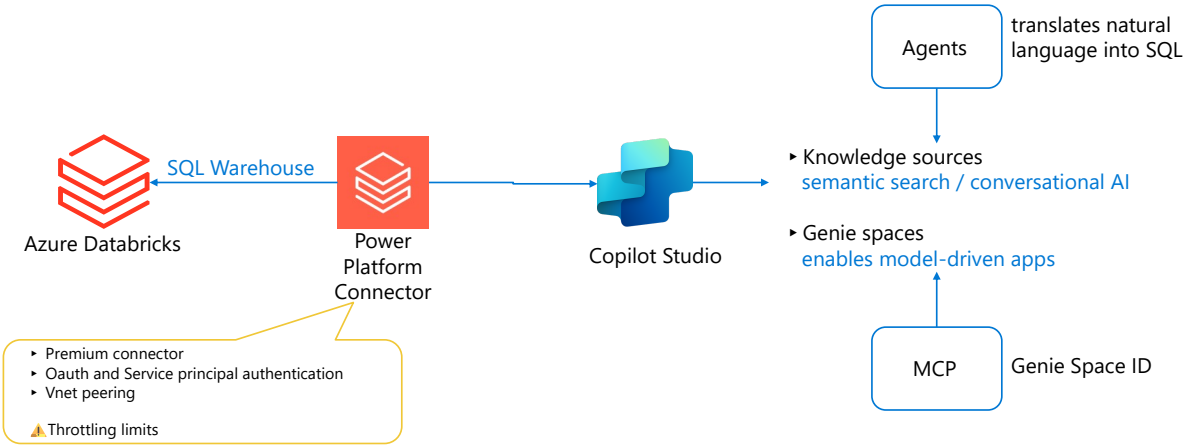
6. Understand integration with Copilot Studio

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-integrations/6-integration-databricks-copilot-studio

Understand integration with Copilot Studio

Completed

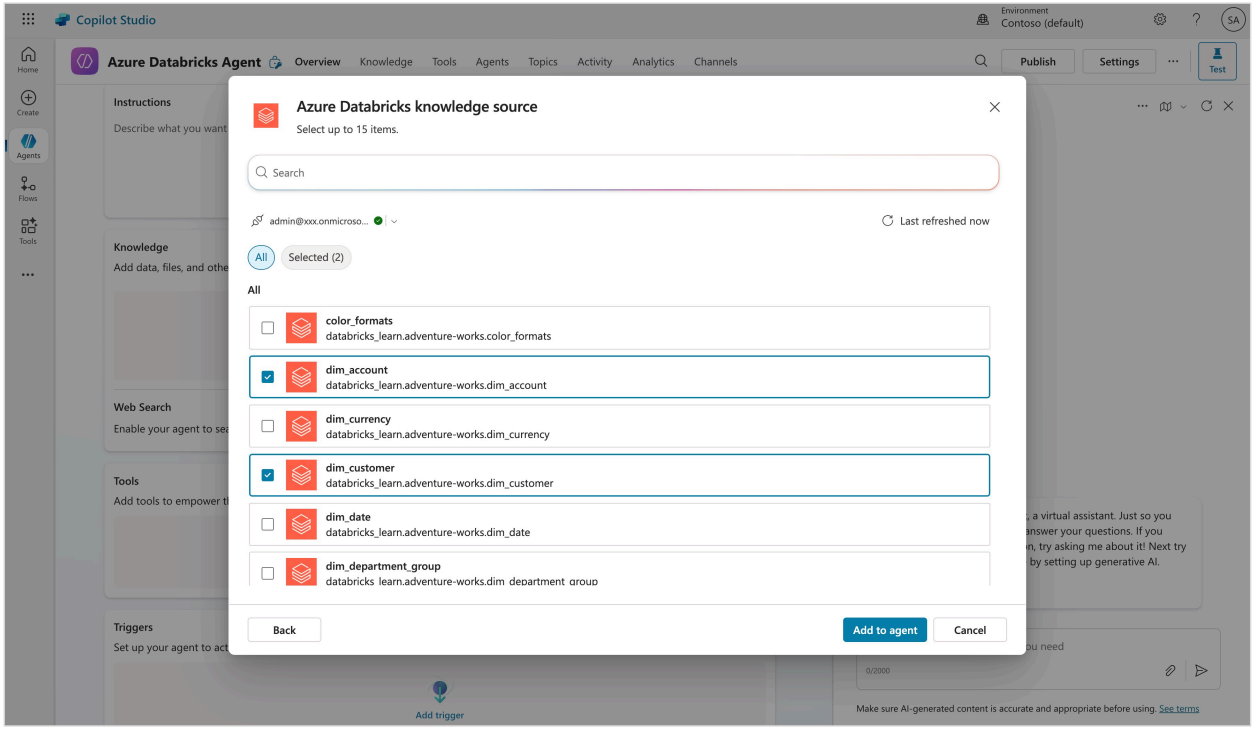
Azure Databricks integrates with Microsoft Copilot Studio to bring governed lakehouse data into conversational AI agents. The **Azure Databricks connector** enables two integration patterns: **knowledge sources** for semantic search and question answering, and **Genie spaces** as intelligent analytical tools. Both approaches preserve Unity Catalog governance, ensuring users access only authorized data.



Use tables as knowledge sources

Copilot Studio agents access Azure Databricks tables as **knowledge sources** to ground responses in your organization's data. Users ask questions in natural language, and the agent retrieves relevant information to provide data-driven answers.

Configure knowledge sources by connecting Copilot Studio to Azure Databricks using OAuth authentication or a service principal. Select a catalog and specific tables for your agent. The agent queries tables directly through the SQL warehouse without copying data, ensuring Unity Catalog enforces all governance controls including row-level security and column masking.



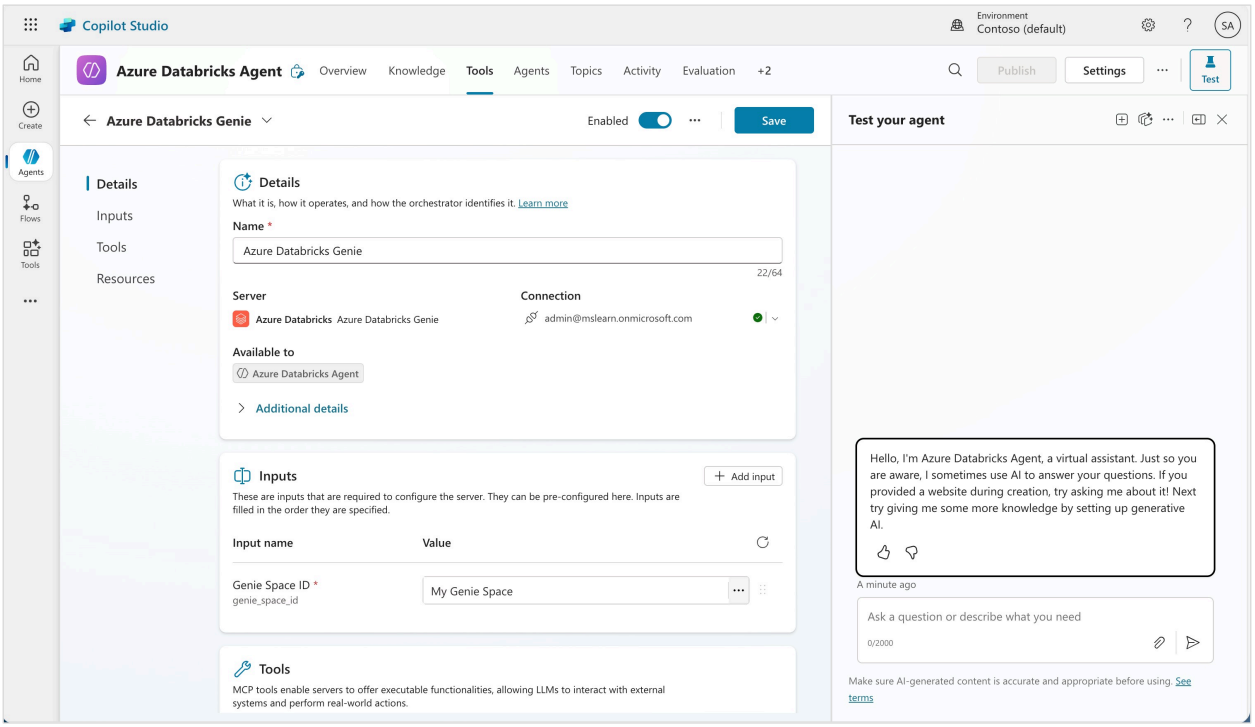
OAuth authentication passes each user's identity to Azure Databricks, where Unity Catalog evaluates permissions in real time. Service principal authentication runs all queries with consistent permissions, useful for uniform data access or cross-tenant scenarios.

The agent performs **semantic search** by translating natural language into SQL, retrieving relevant rows, and synthesizing conversational responses. This enables business users to explore data without writing SQL or understanding the data model.

> **Note**
>
> Knowledge sources work best with tables that have descriptive column names and well-structured data. Consider creating curated views or aggregated tables specifically for agent consumption.

Connect Genie spaces as tools

Copilot Studio agents can interact with **Genie spaces** as intelligent tools. Genie is Azure Databricks' AI-powered analytics interface that understands natural language questions. The integration uses **Model Context Protocol (MCP)**, enabling your agent to orchestrate between conversation management and analytical queries.



To set up Genie integration, enable **Managed MCP Servers** preview in your Azure Databricks workspace, then connect Genie in Copilot Studio using the Genie Space ID. The agent can then send analytical questions to Genie and return results conversationally.

Genie differs from knowledge sources: instead of semantic search and data retrieval, Genie performs complex analytics, generates visualizations, aggregates data, and explains trends through its specialized analytics engine.

> **Important**
>
> Genie integration is currently in Public Preview and requires enabling the Managed MCP Servers preview in your workspace. Not all Azure Databricks features in preview are recommended for production workloads.

Integration scenarios

Customer support agents answer questions about order history, inventory, or shipping status using Azure Databricks tables as knowledge sources. Unity Catalog ensures each representative sees only authorized orders and customers.

Executive analytics agents connect to Genie spaces for business insights. When asked "What drove the increase in customer churn last quarter?", the agent invokes Genie to analyze trends, identify factors, and present findings conversationally.

Data literacy agents help employees understand datasets by answering questions like "What tables contain customer demographic data?" or "How is revenue calculated in the sales fact table?".

Compliance and audit agents query Unity Catalog metadata to answer questions about data lineage, access patterns, or governance policies, helping teams respond efficiently to audit requests.

These scenarios combine conversational AI with governed data access—users interact through natural language while Unity Catalog enforces security, audit logging, and access controls.

Considerations and limitations

The connector **doesn't support government clouds** (GCC, GCC High, China Cloud).

Agent performance depends on **table structure and data quality**. Use clear column names, consistent data types, and appropriate indexes. Consider creating dedicated views for agent consumption rather than exposing complex schemas with many joins.

The connector requires a **SQL warehouse**—serverless or pro only (classic warehouses aren't supported for Genie). Size appropriately for many small queries and configure auto-scaling and auto-stop for cost optimization.

Genie integration is in Public Preview. Enable Managed MCP Servers in workspace settings and consider stability requirements before production deployment.

Service principal authentication executes all queries with the principal's permissions, providing consistent access but preventing user-level row security. OAuth enables user-specific governance but requires users to have Azure Databricks permissions and both platforms in the same Microsoft Entra tenant.

Power Platform **throttling limits** apply: 100 API calls per 60 seconds. Design agents to batch operations and cache frequently accessed information.

7. Understand integration with Microsoft Purview

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-integrations/7-integration-databricks-microsoft-purview

Understand integration with Microsoft Purview

Completed

As organizations build data lakehouses in Azure Databricks, they face data governance challenges. Without integration between Databricks and governance tools, data engineers, business users, and

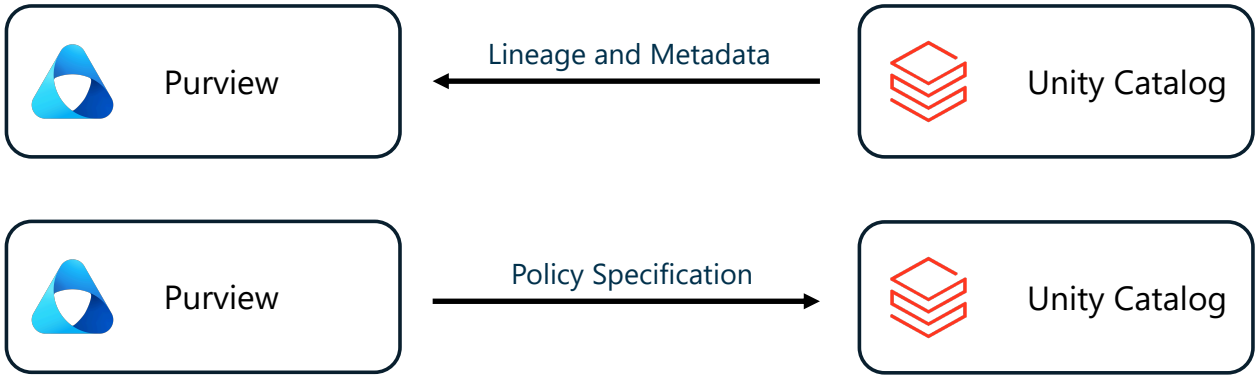
compliance teams work in isolation, creating data silos and governance gaps.

Microsoft Purview provides unified data governance across multiple sources, including Azure Databricks. This integration enables a single catalog of data assets, lineage tracking, and consistent governance policies.

How the integration works

The integration uses **metadata synchronization**. Purview reads metadata about databases, tables, columns, and relationships from Azure Databricks without accessing actual data. This metadata is cataloged alongside other sources.

The process involves **registration** (establishing the connection and authentication) and **scanning** (extracting metadata at scheduled intervals or on demand).



Scanning metadata sources

Purview can scan metadata from both Hive Metastore and Unity Catalog, each offering different capabilities based on the architectural differences between these metadata storage systems.

Scanning Hive Metastore metadata

Purview scans workspace-scoped Hive metastores to discover databases, tables, views, and column definitions. For external tables, it captures **storage relationships** between logical tables and physical storage locations.

Hive metastore scanning captures **static lineage** from view definitions, showing dependencies between views and underlying tables. This requires a self-hosted integration runtime to connect to Azure Databricks clusters.

> **Note**
>
> Hive Metastore scanning does not support incremental scanning. Each scan performs a full extraction, unlike Unity Catalog which supports incremental synchronization.

Scanning Unity Catalog metadata

Unity Catalog scanning extracts the complete hierarchy—metastores, catalogs, schemas, tables, and views. It supports **incremental synchronization**, processing only changes after the initial full scan, which improves efficiency for large environments.

Unlike Hive metastore, Unity Catalog scanning uses cloud-native Azure integration runtime to connect through SQL Warehouses, simplifying deployment.

Understanding runtime lineage

Unity Catalog scanning captures **runtime lineage**—actual data transformations from notebook execution. When notebooks read, transform, and write data, Unity Catalog records these operations in system tables that Purview scans.

Runtime lineage provides:

- **Table lineage**: shows which tables feed into other tables
- **Column lineage**: traces specific columns from source to destination

Limitations: Only transformations logged by Unity Catalog appear in lineage. External orchestration tools like Azure Data Factory don't appear, and column-level lineage may not capture all complex transformations.

Benefits of integration

Unified data discovery

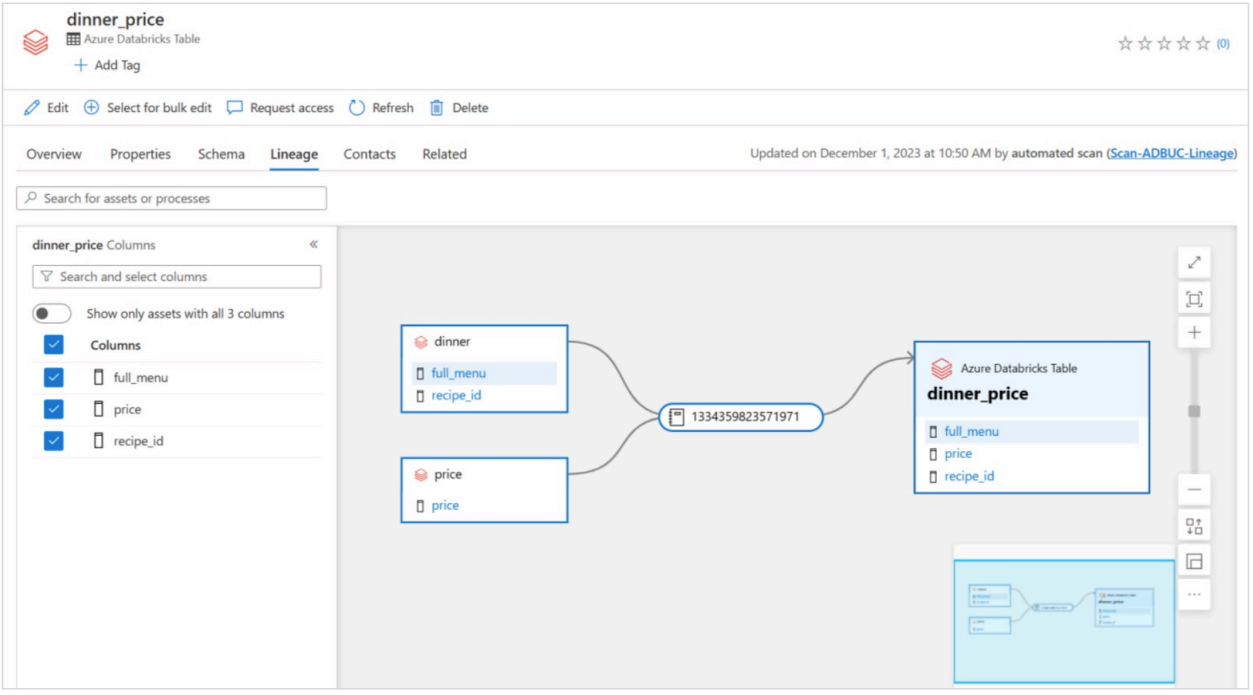
Integration eliminates catalog fragmentation. Azure Databricks tables appear in Purview alongside other data sources, enabling users to find all relevant data regardless of origin. This reduces duplication and improves collaboration.

Consistent governance

Governance teams apply classifications, ownership, and policies consistently across all data sources. Data classification and compliance reporting span the entire data estate using unified tools and processes.

Lineage for impact analysis and compliance

Lineage enables **impact analysis** by showing which downstream assets depend on source data before making changes. For **compliance**, lineage automatically documents data flows required by certain regulations, transforming manual documentation into an automated process.



Scanning approaches

Organizations choose between Hive Metastore and Unity Catalog scanning based on their architecture and governance maturity. During **migration** periods, Purview can scan both simultaneously without duplicating assets.

The approaches differ in infrastructure: Hive Metastore requires self-hosted infrastructure, while Unity Catalog uses cloud-native Azure services through SQL Warehouses. Choose based on network requirements and cloud-native preferences.

8. Understand integration with Microsoft Foundry

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-integrations/8-integration-databricks-microsoft-foundry

Understand integration with Microsoft Foundry

Completed

Azure Databricks and Microsoft Foundry serve complementary roles in modern AI and data architectures. Azure Databricks provides a platform for large-scale data engineering, transformation,

and machine learning, while Microsoft Foundry offers tools for building and deploying AI agents and generative AI applications. Understanding how these platforms integrate helps you design solutions that combine enterprise data capabilities with conversational AI experiences.

The integration between Azure Databricks and Microsoft Foundry uses the **Model Context Protocol (MCP)** to connect AI agents with data insights from Azure Databricks **Genie spaces**. This connection lets AI agents in Microsoft Foundry query data managed in Azure Databricks using natural language, without requiring direct database access or custom API development.

What is the Azure Databricks connector in Microsoft Foundry

Microsoft Foundry supports adding external tools and services to AI agents through standardized connectors. The **Azure Databricks Genie connector** appears as a tool option within Microsoft Foundry, enabling AI agents to access curated datasets and business intelligence insights from Azure Databricks.

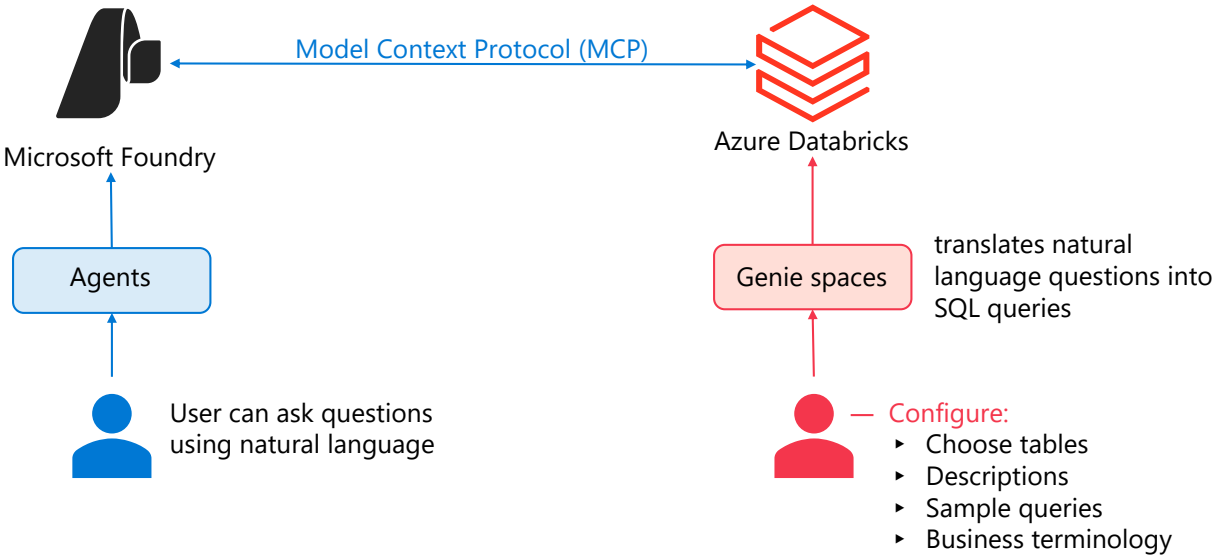
The connector works through Genie spaces, which are conversational interfaces built on top of Azure Databricks data. Domain experts configure these spaces with specific datasets, sample queries, and business terminology. When you add a Genie space as a tool in Microsoft Foundry, your AI agents gain the ability to answer questions about that data using natural language.

With this connector, Microsoft Foundry agents don't query raw tables or write SQL directly. Instead, they interact with the preconfigured Genie space, which translates natural language questions into SQL queries and returns results. This approach provides a layer of abstraction that simplifies integration while maintaining data governance through the Genie space configuration.

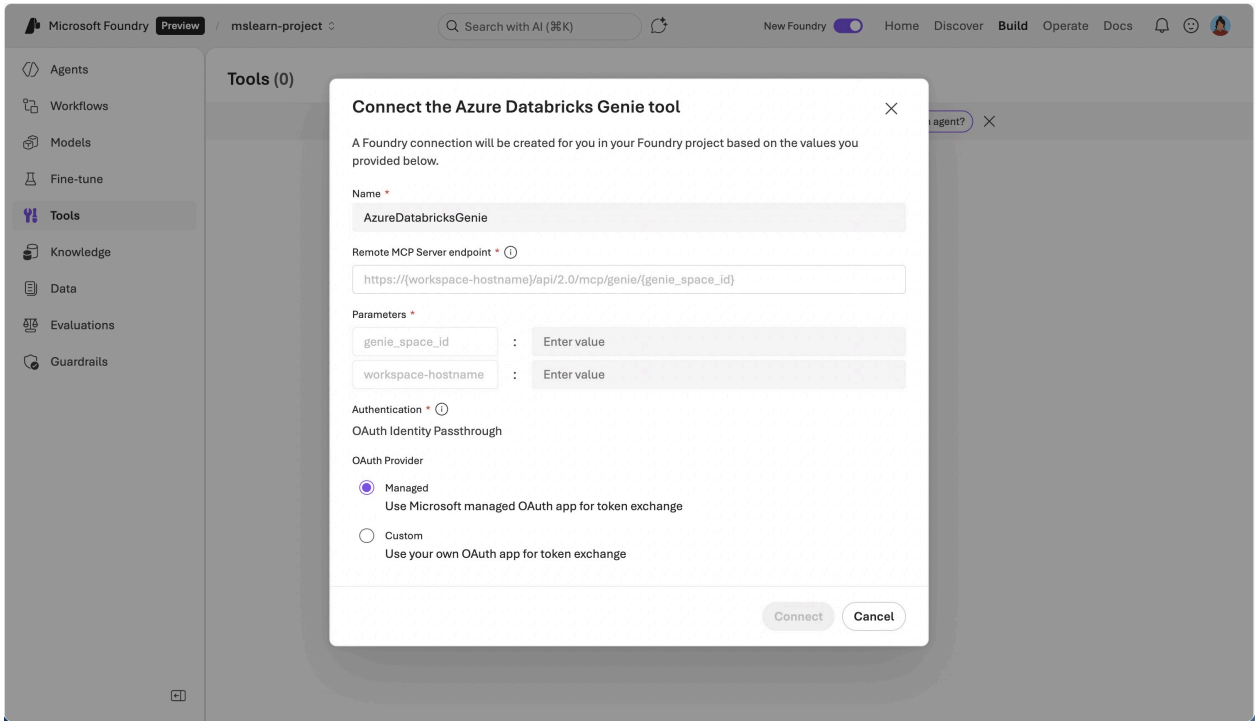
The integration relies on the Model Context Protocol, an open standard that defines how AI agents communicate with external tools and data sources. Azure Databricks implements MCP through managed servers that expose Genie spaces, Unity Catalog functions, and other resources to compatible AI platforms. Microsoft Foundry acts as an MCP client, sending requests to the Azure Databricks MCP server and receiving structured responses.

How the integration works

The architecture involves three main components working together. **Microsoft Foundry** hosts the AI agent and manages user interactions. The **Azure Databricks MCP server** exposes Genie spaces as MCP-compatible tools. **Genie spaces** contain the curated data, metadata, and business logic that enable natural language querying.



When you connect a Genie space to a Microsoft Foundry agent, you specify connection details including your workspace hostname, Genie space ID, and authentication method. Microsoft Foundry uses these details to establish a secure connection to the Azure Databricks MCP server endpoint. The MCP server validates permissions and provides access to the specific Genie space you've configured.



During runtime, when a user asks the AI agent a question, Microsoft Foundry determines whether to route that question to the Genie tool based on the agent's configuration and the question's context. If the Genie tool is invoked, Microsoft Foundry sends the question to the Azure Databricks MCP server. The Genie space processes the question, generates a SQL query, executes it against the underlying data, and returns the results to Microsoft Foundry. The AI agent then incorporates these results into its response to the user.

Authentication occurs through **OAuth Identity Passthrough**, which means user identities flow from Microsoft Foundry to Azure Databricks. You can choose between **Managed** OAuth, which uses Microsoft Entra ID authentication, or **Custom** OAuth for other identity providers. Databricks recommends Managed OAuth for most scenarios. Each time the AI agent accesses the Genie space, Azure Databricks validates that the requesting identity has appropriate permissions to use that space.

When to use this integration

This integration enables several practical scenarios where combining conversational AI with enterprise data creates value. You can build **AI assistants** that answer business questions using live data from your lakehouse. Users interact with these assistants through natural language, and the assistants query Databricks data through Genie spaces to provide accurate, data-driven responses.

For **self-service analytics**, business users can ask questions about data without learning SQL or understanding database schemas. The Genie space handles the complexity of translating questions into queries, while the Microsoft Foundry agent provides a conversational interface. This pattern works well when you want to extend data access beyond technical teams while maintaining governance through Genie space configuration.

You might use this integration to build **domain-specific assistants** for different parts of your organization. For example, a sales team assistant could connect to a Genie space configured with sales pipeline data, while a finance assistant connects to a different Genie space with financial metrics. Each assistant operates within its defined data scope, and you control what data each Genie space exposes.

The integration also supports **multi-tool agents** that combine Databricks data insights with other capabilities. An AI agent in Microsoft Foundry might use the Genie connector to retrieve data, use another tool to perform calculations or predictions, and use additional tools to take actions based on the results. This orchestration happens within Microsoft Foundry's agent framework, while Azure Databricks provides the data foundation.

Key considerations

Security and permissions require attention at multiple levels. In Azure Databricks, you must enable the **Managed MCP Servers** preview feature in your workspace. Users who configure the connection in Microsoft Foundry need permissions to use the Genie space in Azure Databricks. The Genie space itself enforces permissions on the underlying data through Unity Catalog, so users can only access data they're authorized to see through the Genie space configuration.

Genie space setup directly impacts the quality of responses your AI agent can provide. Domain experts should configure Genie spaces with relevant datasets, clear column descriptions, and sample queries that reflect common questions. The better configured your Genie space, the more effectively it can translate natural language questions into accurate SQL queries. Consider this as a one-time setup cost that improves the ongoing experience for all users.

Performance depends on both Microsoft Foundry and Azure Databricks. Questions that require complex queries or access large datasets take longer to process than simple lookups. The underlying SQL warehouse or compute resources in Azure Databricks affect query execution speed. Plan your compute capacity based on expected query patterns and user concurrency.

The integration has **specific limitations** to consider. The Genie MCP server doesn't maintain conversation history when invoked as an MCP tool, so each question is processed independently. If your scenario requires multi-turn conversations with context, you might need to explore alternative integration patterns. Additionally, network access between Microsoft Foundry and your Azure Databricks workspace must be configured appropriately, considering any IP restrictions or private endpoints you've implemented.

Cost considerations include both Microsoft Foundry and Azure Databricks components. Microsoft Foundry charges for agent usage and API calls. Azure Databricks charges for serverless SQL compute when Genie spaces execute queries. Genie uses serverless SQL compute to run, and pricing follows the standard Databricks SQL serverless pricing model. Evaluate your expected query volume and complexity to estimate costs accurately.

> **Note**
>
> The feature is currently in **Public Preview**, which means it's suitable for development and testing but may undergo changes before general availability. Plan for potential updates to configuration requirements or functionality as the feature evolves. Monitor Azure Databricks release notes for updates that affect your implementation.

9. Module assessment

https://learn.microsoft.com/en-us/training/modules/understand-azure-databricks-integrations/9-knowledge-check

Module assessment

Completed

10. Summary

Summary

Completed

Azure Databricks becomes more powerful when integrated with the Microsoft ecosystem. Throughout this module, you explored seven integrations that extend your data lakehouse capabilities across analytics, development, applications, and AI services. Each integration serves specific use cases while maintaining Unity Catalog as the central governance layer.

You learned how Microsoft Fabric and Power BI enable business intelligence, letting analysts create reports from Databricks data. The Visual Studio Code integration transforms development by enabling local coding with remote execution and debugging. Power Platform and Copilot Studio bring governed data into applications and AI agents, making lakehouse data accessible through familiar interfaces. Microsoft Purview provides unified governance, while Microsoft Foundry connects AI agents with analytical insights.

Understanding the security models, authentication methods, and limitations of each integration helps you design solutions that balance accessibility with governance. As you implement these patterns, match integration choices to specific business scenarios, evaluate security implications carefully, and test configurations thoroughly. The combination of Azure Databricks with Microsoft services creates comprehensive data and AI solutions that serve diverse needs while maintaining centralized governance and quality standards.

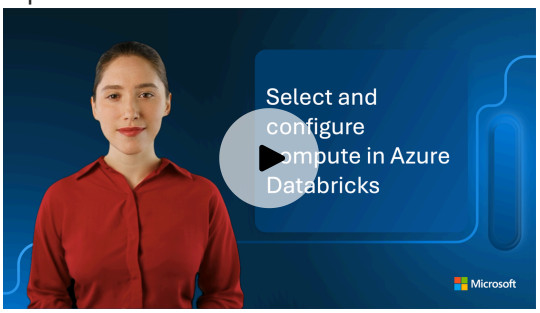
Select and Configure Compute in Azure Databricks

1. Introduction

https://learn.microsoft.com/en-us/training/modules/select-and-configure-compute/1-introduction

Introduction

Completed



Every Azure Databricks workload runs on compute resources, but choosing the wrong compute type or configuration leads to unnecessary costs, poor performance, or blocked functionality. **Serverless compute** starts in seconds but doesn't support RDD APIs. **Classic compute** offers complete flexibility but requires more management overhead. **SQL warehouses** excel at analytical queries while **job clusters** optimize for automated workflows. Understanding these differences helps you match compute to workload requirements.

Beyond selecting a compute type, configuration decisions shape how your workload performs. **Node types** determine processing capacity and memory availability. **Autoscaling** balances cost and responsiveness. **Access permissions** control who can use compute resources while **library installations** provide the dependencies your code needs. Each configuration choice affects multiple dimensions—performance, cost, security, and operational complexity.

Getting compute configuration right matters throughout the development lifecycle. During exploration, you need fast iteration cycles and minimal setup overhead. In production, stability and cost efficiency take priority. **Photon acceleration** can double query performance for SQL workloads. **Instance pools** reduce startup time but require paying for idle capacity. **Dedicated access modes** enable secure group collaboration while enforcing permission boundaries.

In this module, you learn to evaluate compute options systematically and configure resources that match your workload characteristics. You discover when serverless compute provides the best experience, how to tune performance settings for different scenarios, and best practices for managing access and dependencies across your organization.

2. Choose an appropriate compute type

https://learn.microsoft.com/en-us/training/modules/select-and-configure-compute/2-choose-appropriate-compute-type

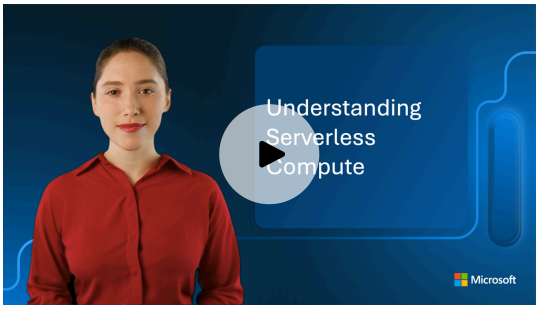
Choose an appropriate compute type

Completed

When you work with Azure Databricks, selecting the right **compute type** directly affects your costs, performance, and operational complexity. Each compute option serves different workload patterns and comes with distinct tradeoffs in startup time, scalability, and management overhead.

Understanding these options helps you match compute resources to your specific needs. You can optimize for fast development cycles, minimize costs for production jobs, or balance both for analytics workloads.

Serverless compute



Serverless compute is managed entirely by Azure Databricks. You don't provision or configure infrastructure—Azure Databricks automatically allocates and scales resources based on your workload demands. These resources run in **Databricks' Azure subscription, not yours**, which means no virtual machines or networking components appear in your subscription.

With serverless compute, startup typically takes 2-6 seconds. The platform scales up rapidly when query volume increases and scales down during idle periods to minimize costs. This eliminates the need to estimate capacity or manage cluster configurations.

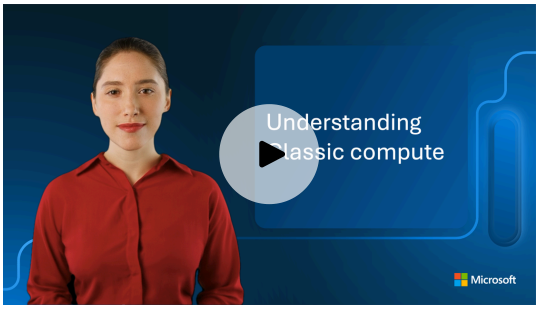
Serverless compute requires **Unity Catalog** and is available for:

- **Notebooks**: Interactive Python and SQL development with automatic resource allocation
- **Jobs**: Automated workflows that run without infrastructure setup
- **Pipelines**: Lakeflow Spark Declarative Pipelines with on-demand scaling
- **SQL warehouses**: Optimized SQL query execution with intelligent workload management

Serverless works best for exploratory analysis, ETL pipelines, business intelligence workloads, and scenarios where startup latency matters. The versionless runtime means Azure Databricks automatically applies upgrades, so you always run on the latest features without migration effort.

However, serverless has limitations. You can't use **RDD APIs** (Resilient Distributed Dataset), **R language**, **JAR libraries** in notebooks, or custom Spark configurations. If your workload requires these capabilities, consider classic compute instead.

Classic compute



Classic compute gives you full control over cluster configuration. You create, size, and manage compute resources that run directly in **your Azure subscription**, giving you visibility and control over the underlying infrastructure.

Classic compute supports two access modes that determine how users interact with the cluster:

Standard access mode enables multiple users to share a single cluster concurrently. Lakeguard provides isolation between user code, preventing one user's work from interfering with another's. This mode works well for collaborative data engineering, shared analytics, and cost optimization through resource pooling.

Dedicated access mode assigns the cluster exclusively to a single user or group. With dedicated access, you get full machine-level privileges, which you need for RDD APIs, GPU workloads, R language support, or custom container environments.

Beyond access modes, classic compute offers different **cluster modes** that determine the cluster architecture:

Multi-node clusters consist of one driver node and one or more worker nodes. The driver coordinates execution while workers perform distributed computations in parallel. This architecture enables horizontal scaling—you can add worker nodes to process larger datasets or increase

parallelism. Multi-node clusters work well for production workloads that process large volumes of data or require high throughput.

Single-node clusters contain only a driver node with no worker nodes. All computation happens on the driver, which means workloads can't distribute across multiple machines. Single-node clusters suit specific scenarios like lightweight exploration, small dataset analysis, or machine learning experimentation where shuffle overhead would exceed the benefits of distribution. They're particularly useful for ML workflows with frameworks like scikit-learn that don't inherently distribute across nodes, or when testing notebooks with small data samples.

Keep in mind that single-node clusters have architectural limitations. They can't scale horizontally to handle increased load, and all processing relies on the resources of a single machine. For workloads requiring distributed processing, parallel task execution across large datasets, or fault tolerance through redundancy, use multi-node clusters instead.

Classic compute offers flexibility but requires more management. You configure instance types, autoscaling rules, and runtime versions. Startup time typically ranges from 3-7 minutes depending on cluster size. Unlike serverless, you select and manage the Databricks Runtime version yourself—choosing when to upgrade from one version to another (for example, from Runtime 13.3 LTS to 14.3 LTS). While underlying OS and security updates can be automated, runtime version changes require manual selection.

This compute type fits workloads that need features unavailable in serverless, require precise control over infrastructure, or have compliance requirements for resource isolation.

SQL warehouses

SQL warehouses are compute resources optimized specifically for SQL queries, analytics, and business intelligence. They come in three types, each with different performance characteristics.

Serverless SQL warehouses offer optimal performance and cost efficiency. They start in 2-6 seconds, use Intelligent Workload Management to predict query resource needs, and scale clusters dynamically based on demand. Photon and Predictive IO accelerate query execution. Choose serverless SQL warehouses for most SQL workloads—BI dashboards, ETL jobs, and ad hoc analysis.

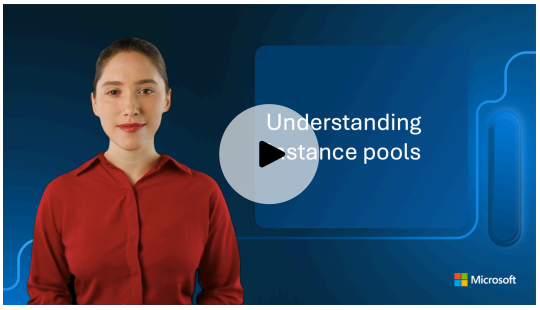
Pro SQL warehouses support Photon and Predictive IO but lack Intelligent Workload Management. They take approximately 4 minutes to start and scale less dynamically than serverless. You use pro

warehouses when you need custom networking configurations—for example, connecting to on-premises databases through federation or integrating with services in your virtual network.

Classic SQL warehouses offer entry-level SQL performance with Photon support only. They start in about 4 minutes and include basic autoscaling. Choose classic warehouses only when serverless and pro options aren't available or for basic interactive exploration with minimal performance requirements.

All SQL warehouse types optimize for SQL execution patterns, but serverless offers the most responsive scaling and lowest operational overhead.

Instance pools



Instance pools maintain a set of idle virtual machine instances ready for immediate use. When you create a cluster from a pool, startup time decreases because Databricks allocates instances from the pool instead of requesting new ones from Azure.

Pools reduce startup time from minutes to under a minute in many cases. You configure the minimum number of idle instances to keep warm and the maximum pool capacity. When clusters release instances, those instances return to the pool for reuse.

You pay for virtual machine costs while instances sit idle in the pool, but not for Azure Databricks compute units. This makes pools cost-effective when you run workloads frequently enough that the reduced startup time justifies the idle infrastructure cost.

With serverless compute available, pools matter less for most scenarios. Serverless starts faster and scales more efficiently without requiring you to maintain idle capacity. However, pools remain useful when you need classic compute features and want to optimize startup time for frequently run workloads.

Configure pools with spot instances for worker nodes to reduce costs, but use on-demand instances for driver nodes to maintain reliability.

Job compute

Job compute refers to clusters optimized for automated workflows rather than interactive development. You configure job compute through cluster policies that enforce best practices for production workloads.

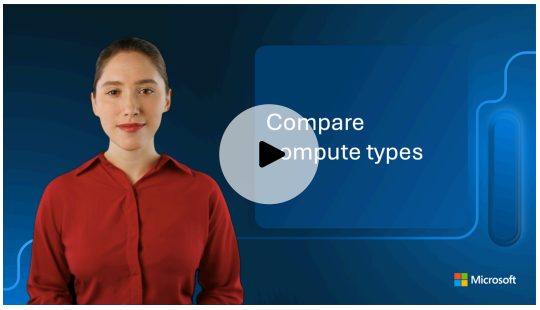
Job clusters terminate automatically after completing their tasks, preventing unnecessary costs from idle resources. When you configure a job, you choose between serverless and classic job compute.

- **Serverless job compute** offers faster startup, automatic infrastructure management, and lower costs for most automated workloads.
- **Classic job compute** provides more configuration options for workloads that need features serverless doesn't support.

With classic job compute, you can use optimized settings like autoscaling and **spot instances**. **Spot instances** (also called Azure Spot VMs) use Azure's excess capacity at significantly reduced costs—often up to 90% cheaper than regular on-demand instances. Azure can reclaim these instances with only 30 seconds notice when it needs the capacity back. Spark's built-in fault tolerance automatically handles these interruptions by retrying failed tasks on other available nodes, which makes spot instances a viable cost-saving option for many batch processing and ETL workloads.

The Job Compute policy in Azure Databricks offers a template for creating production-ready job clusters with sensible defaults. It enforces the latest LTS (Long Term Support) runtime version and other reliability settings.

Compare compute types

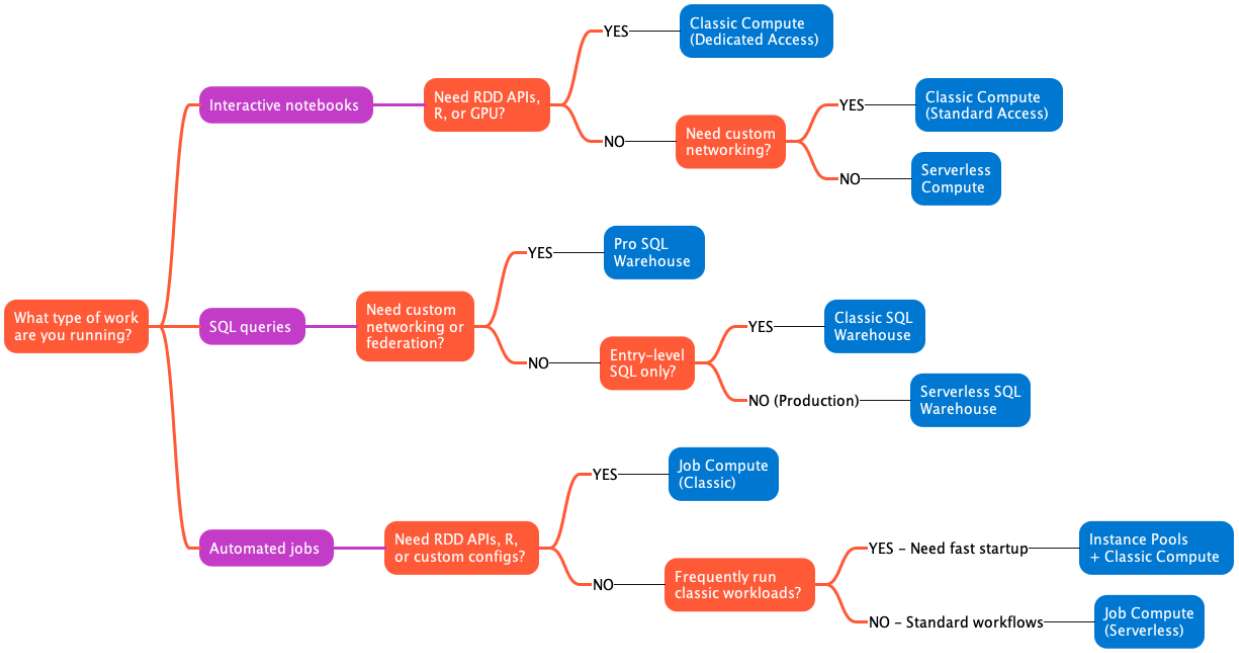


Different compute types suit different scenarios. The following table compares key characteristics to help you make informed decisions:

Compute type	Recommended for	Startup time	Management overhead	Cost efficiency	Key limitation
Serverless compute	Interactive development, ETL jobs, BI workloads	⚡ 2-6 seconds	👌 Minimal - fully managed	🟢 High - scales to zero, pay only for usage	No RDD APIs, R, or JAR libraries
Classic compute (Standard)	Collaborative data engineering, shared analytics	⏱️ 3-7 minutes	🔧 Moderate - configure and monitor	🟡 Moderate - multi-user sharing reduces per-user cost	Requires Unity Catalog for governance
Classic compute (Dedicated)	RDD workloads, GPU jobs, R language, custom containers	⏱️ 3-7 minutes	🔧 Moderate - configure and monitor	🔴 Lower - single user/group only	Higher cost than shared resources
Instance pools	Frequently run classic workloads needing fast startup	🚀 <1 minute	⚠️ Higher - maintain idle capacity	🔄 Variable - justify idle cost with usage frequency	Pay for idle instances
SQL warehouse (Serverless)	SQL analytics, BI dashboards, reporting	⚡ 2-6 seconds	👌 Minimal - intelligent workload management	🟢 High - dynamic scaling with Photon + Predictive IO	SQL workloads only
SQL warehouse (Pro)	SQL with custom networking, federation	⏱️ ~4 minutes	🔧 Moderate - manual scaling configuration	🟡 Moderate - Photon + Predictive IO	Slower scaling than serverless
SQL warehouse (Classic)	Entry-level SQL exploration	⏱️ ~4 minutes	🔧 Moderate - manual scaling configuration	🔴 Lower - basic Photon only	Limited performance features
Job compute (Serverless)	Automated workflows, production ETL	⚡ 2-6 seconds	👌 Minimal - auto-terminated after completion	🟢 High - no idle costs	Same as serverless compute
Job compute (Classic)	Jobs requiring custom configurations	⏱️ 3-7 minutes	🔧 Moderate - configure policies	🟡 Moderate - auto-termination prevents idle waste	Requires infrastructure management

Choose the right compute type

Start your decision-making process by identifying your workload characteristics. The following diagram illustrates a decision flow to help you select the appropriate compute type:



Consider these questions:

What type of work are you running? If you're writing interactive notebooks, serverless compute offers the fastest iteration cycle. For SQL queries and BI dashboards, serverless SQL warehouses deliver optimal performance. Automated production jobs work well with serverless job compute unless you need custom configurations.

Does your code use specific APIs or languages? RDD APIs, **R language**, or **GPU acceleration** require classic compute with dedicated access mode. Python, SQL, and Scala workloads run on either serverless or standard classic compute.

How frequently does this workload run? Infrequent workloads benefit most from serverless because you pay only during execution. Recurring workloads might justify instance pools if you use classic compute, though serverless often still provides better economics.

Do you need custom networking or specific infrastructure? Custom virtual networks, on-premises connectivity, or specific instance types require classic compute or pro SQL warehouses. Serverless operates in the Databricks-managed subscription without custom network integration, so resources don't appear in your Azure subscription.

What are your performance requirements? For latency-sensitive workloads, serverless compute starts 2-4x faster than classic options. For predictable performance at scale, both serverless and classic can meet requirements, but serverless adapts more dynamically to load variations.

For most scenarios, start with serverless options. They minimize operational overhead, optimize costs through automatic scaling, and provide the fastest development experience. Switch to classic compute only when you encounter a specific limitation that serverless doesn't support.

3. Configure compute performance

https://learn.microsoft.com/en-us/training/modules/select-and-configure-compute/3-configure-compute-performance

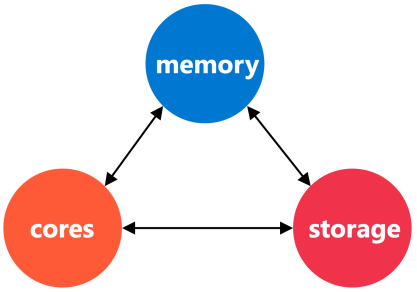
Configure compute performance

Completed

Configuring compute resources involves balancing performance requirements with cost considerations. Over-provisioning leads to unnecessary expenses, while under-provisioning can cause stability issues and slow query execution. Understanding how to configure compute settings helps you optimize resources for your workload.

Understand compute resource components

Compute performance depends on three key factors working together. Each factor influences how efficiently your workload runs and how much it costs.



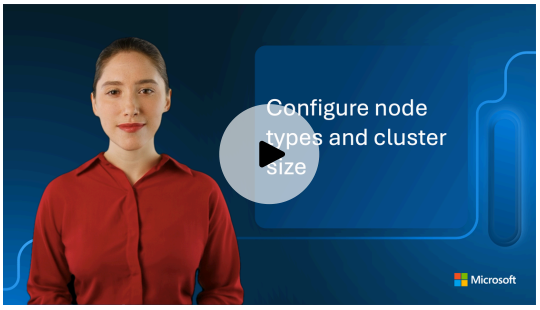
Total executor cores determine the maximum parallelism available for processing data. More cores allow Spark to process more tasks simultaneously. A cluster with 8 workers, each having 4 cores, provides 32 cores total for parallel processing.

Total executor memory affects how much data can be processed in memory before spilling to disk. Memory-intensive operations like joins and aggregations benefit from larger memory configurations. When memory runs out, Spark writes data to disk, which significantly slows performance.

Local storage provides temporary space for shuffle operations and caching. During shuffle operations, Spark writes intermediate data to local disks on worker nodes. Fast local storage reduces the time spent on these operations.

With this understanding of compute components, you can make informed decisions about node types and cluster size.

Configure node types and cluster size



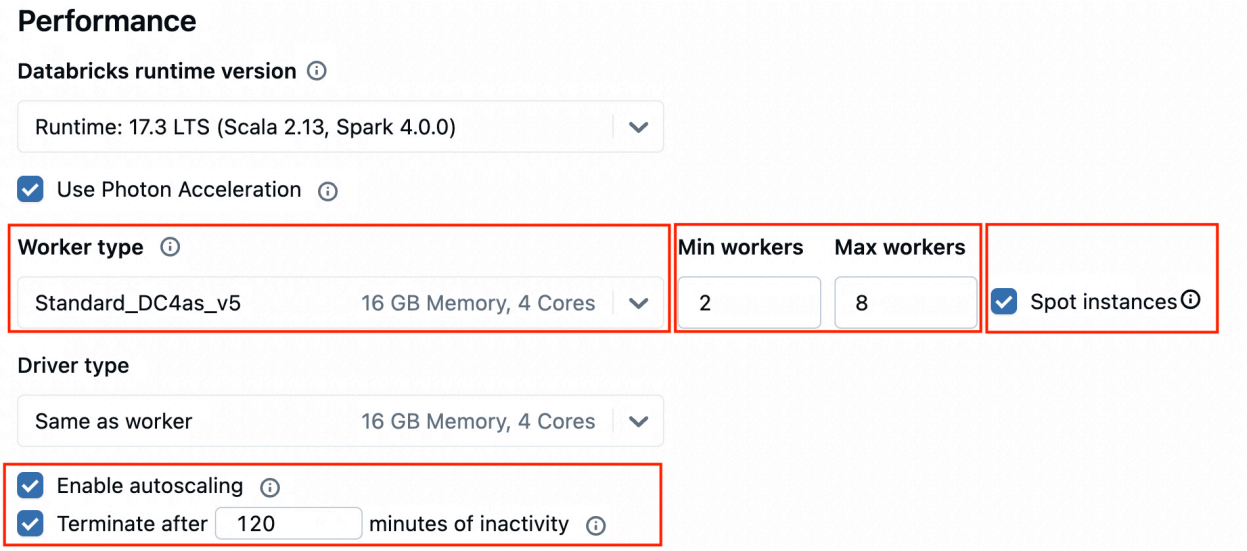
Node type selection directly impacts both performance and cost. Different instance families serve different workload characteristics.

Memory-optimized instances work well for workloads with large **joins**, **aggregations**, or data that needs to stay in memory. These instances provide more RAM per core, reducing the likelihood of spilling data to disk. Examples include **E-series** VMs, which offer high memory-to-core ratios ideal for in-memory analytics.

Compute-optimized instances suit workloads that perform complex **calculations** but don't require large amounts of memory. ETL jobs with straightforward transformations often run efficiently on these instances. Examples include **F-series** VMs, which provide high CPU performance with lower memory ratios.

Storage-optimized instances benefit workloads that repeatedly read the same data or require fast **local disk access**. Data analysis workloads with caching enabled perform better with these instances. Examples include **L-series** VMs, which offer fast local **NVMe storage** for high I/O workloads.

GPU-accelerated instances provide graphics processing units designed for computationally intensive workloads like **machine learning**, **deep learning**, and **image processing**. These instances can accelerate model training by 10-100x compared to CPU-only clusters. Examples include **NC-series** and **ND-series** VMs with NVIDIA GPUs. GPU instances require **Databricks Runtime ML** and work best for tasks like training neural networks, fine-tuning large language models, or running inference on complex models.



The balance between number of workers and instance size affects performance differently depending on your workload. Two workers with 16 cores and 128 GB RAM each provide the same total compute and memory as eight workers with 4 cores and 32 GB RAM each. However, the configuration with **fewer, larger workers** reduces network traffic during shuffle operations, while **more smaller workers** can provide better parallelism for highly distributed workloads.

For analytical workloads with many shuffle operations, **fewer larger workers** typically perform better. For simple batch processing that benefits from high parallelism, **more smaller workers** might be more cost-effective.

Configure autoscaling

Autoscaling adjusts the number of workers based on workload demands, helping you maintain performance while controlling costs.

When you enable autoscaling, you set minimum and maximum worker counts. Azure Databricks monitors workload requirements and adds workers when needed, up to the maximum you specified. When demand decreases, workers are removed down to the minimum.

Azure Databricks uses **optimized autoscaling** by default when you enable autoscaling. Optimized autoscaling scales up quickly in two steps from minimum to maximum. It can scale down even when the cluster isn't idle by monitoring shuffle file state. For **job compute**, it evaluates utilization every 40 seconds. For **all-purpose compute**, it checks every 150 seconds.

Consider autoscaling for workloads with variable resource needs throughout execution. Data exploration sessions often start with small data samples and later process larger datasets. Autoscaling adds workers when you process the larger datasets and removes them when you return to smaller samples.

For predictable workloads that maintain consistent resource usage, a fixed number of workers often provides more stable performance and simpler capacity planning. The overhead of scaling decisions can slightly impact performance for steady-state workloads.

Autoscaling works particularly well with instance pools. Set your minimum workers equal to or less than the minimum idle instances in the pool. This ensures quick scaling because the instances are already provisioned and ready.

Configure termination settings

Automatic termination prevents idle compute resources from accumulating unnecessary costs while maintaining availability for scheduled workloads.

When you configure automatic termination, you specify an inactivity period in minutes. If no commands run on the cluster for longer than this period, Azure Databricks terminates the cluster. The cluster configuration remains available for restart when needed.

For interactive workloads like data analysis, set the termination period based on typical session patterns. A 45-minute timeout works well for most use cases, giving data engineers time to review results between queries without leaving clusters idle for hours.

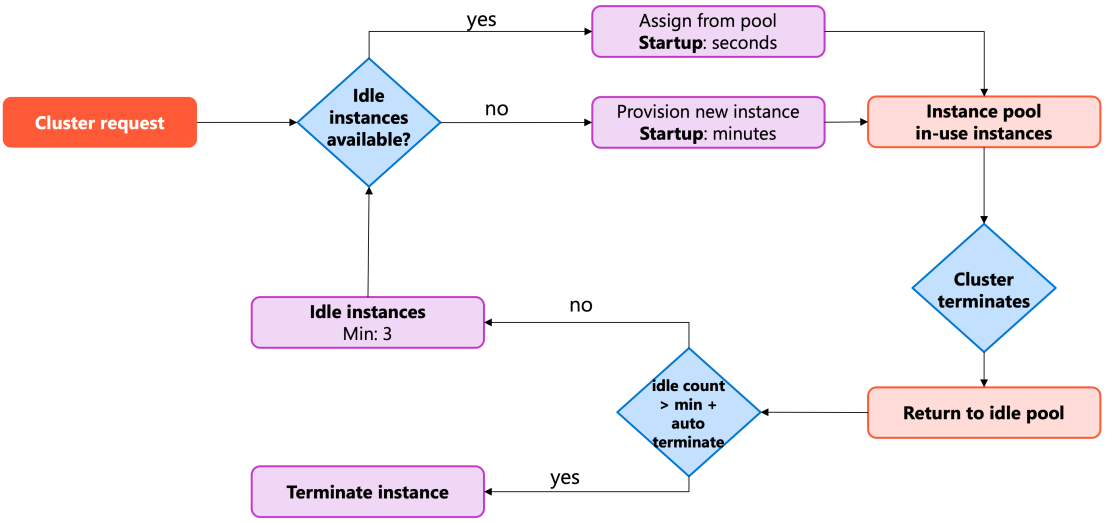
For job compute, automatic termination happens after the job completes. The cluster starts automatically when the next scheduled run begins, so you don't need to manage startup manually.

Spot instances reduce costs but come with availability trade-offs. Azure can reclaim spot instances when capacity is needed elsewhere. For **worker nodes**, spot instances work well because Azure Databricks/Spark can handle worker failures. However, always use **on-demand instances** for **driver nodes**. If the driver is reclaimed, the entire cluster fails.

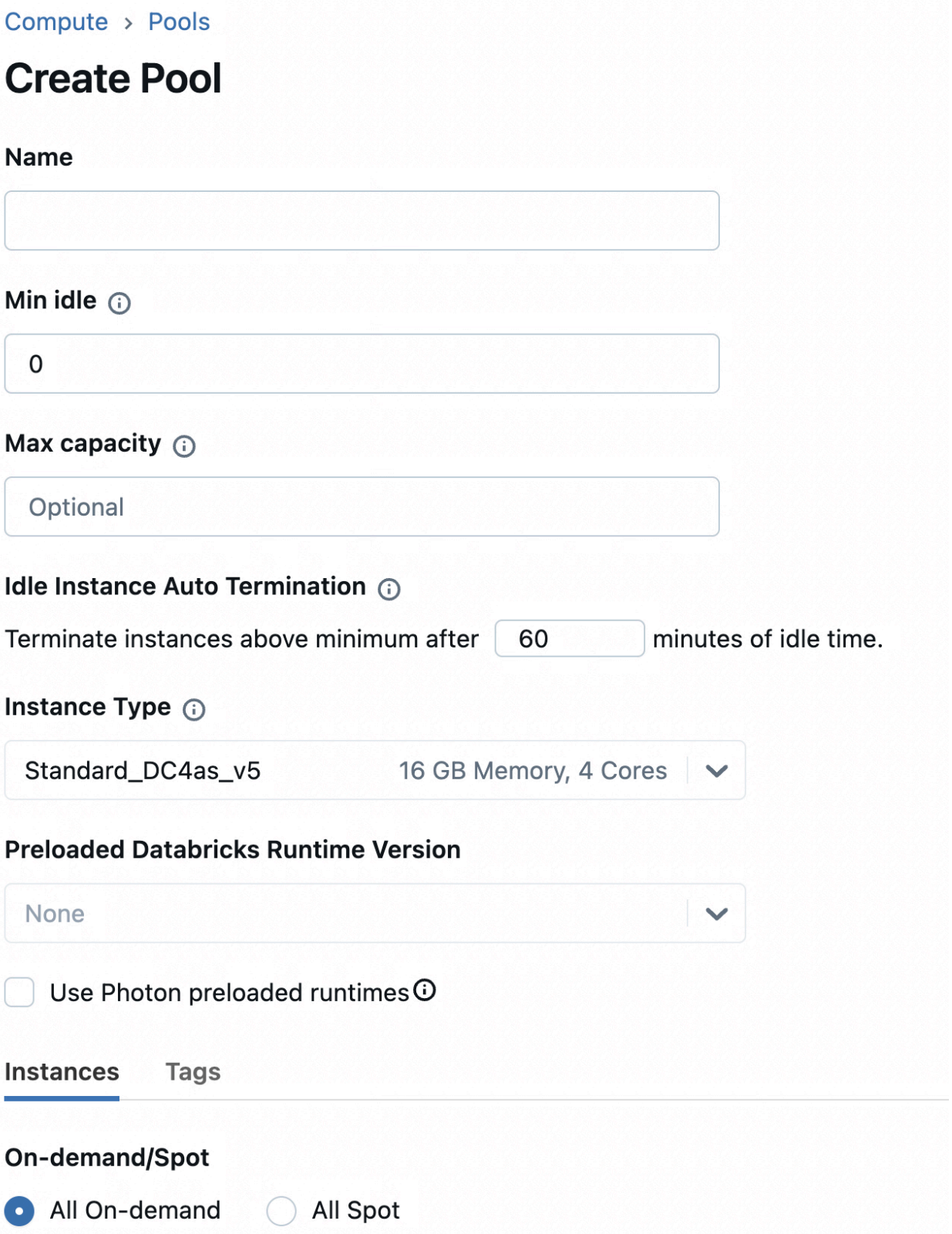
Enable **decommissioning** when using spot instances to reduce task failures. When a spot instance receives a preemption notice, decommissioning migrates shuffle and cached data to healthy workers before the instance terminates. This reduces the need to recompute lost data.

Use instance pools

Instance pools maintain a set of idle instances ready for immediate use, reducing cluster startup time from minutes to seconds.



Configure the **minimum idle instances** to match your typical concurrent cluster needs. If you regularly run three notebooks simultaneously, maintain at least three idle instances. These instances remain available even when not in use, providing instant cluster startup.



Set **maximum capacity** to control costs and prevent one workload from consuming all available resources. When multiple teams share a workspace, pools with maximum capacity settings ensure fair resource distribution. For example, with a 100-instance quota, you might create two pools each with a 50-instance maximum for two teams.

The **idle instance auto termination** setting removes instances that exceed your minimum idle count after the specified period. If you set minimum idle to 3 and auto termination to 30 minutes, a pool that scales up to 8 instances will reduce back to 3 instances after 30 minutes of inactivity.

Preloading a Databricks Runtime version on pool instances accelerates cluster launches even further. When creating a cluster, if you select the preloaded runtime, the cluster starts almost immediately because the runtime is already installed on idle instances.

Pools work best for workloads with frequent cluster creation and termination cycles. Development teams creating and destroying clusters throughout the day see significant time savings. Production jobs that run on dedicated long-running clusters don't benefit as much from pools.

Balance cost and performance

Achieving the right balance between cost and performance requires understanding your workload characteristics and adjusting configurations accordingly.

Start with conservative settings and monitor performance. If you see frequent spilling to disk or slow query execution, increase memory or core count. If utilization remains low, reduce cluster size or enable autoscaling.

> **Note**
>
> Use the **Spark UI** to identify performance issues. Check the **Jobs Timeline** to find long-running stages, and view the **stage details page** for spill statistics showing **Shuffle Spill (Memory)** and **Shuffle Spill (Disk)**. Compare stage durations to identify bottlenecks and slow queries.

Use serverless compute when your workload supports it. Serverless eliminates configuration decisions and automatically scales based on demand, often providing the best cost-performance balance without manual tuning.

Regular monitoring helps you identify optimization opportunities. Review cluster metrics to see actual utilization compared to provisioned capacity. Adjust node types, worker counts, or scaling settings based on observed patterns rather than assumptions.

> **Note**
>
> Monitoring and observability are covered in detail in a later module.

4. Configure compute features

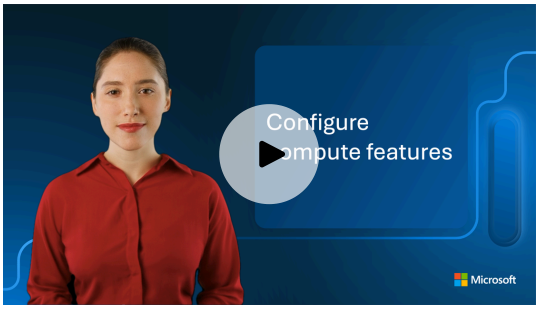
https://learn.microsoft.com/en-us/training/modules/select-and-configure-compute/4-configure-compute-feature-settings

Configure compute features

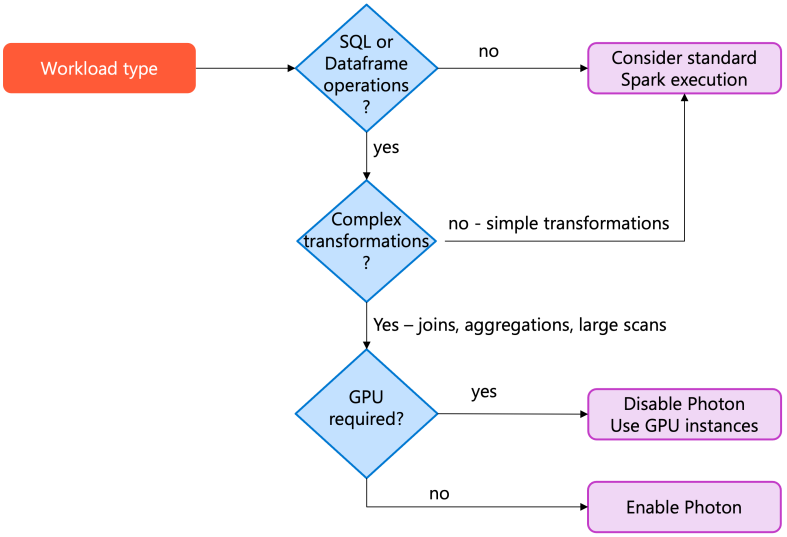
Completed

Compute features determine the functional capabilities available to your workloads. Unlike performance settings that tune resource allocation, feature settings enable specific technologies and runtime environments. Selecting appropriate features ensures your compute resource supports your workload requirements while maximizing efficiency.

Enable Photon acceleration

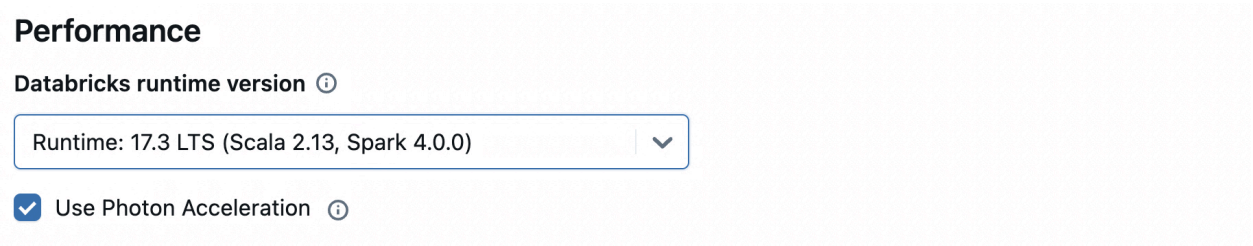


Photon is a query execution engine that replaces traditional Spark components with optimized native code. When you enable Photon, your compute resource uses this accelerated engine for SQL queries and DataFrame operations.



With Photon enabled, queries that involve **complex transformations** run faster. Operations like **joins**, **aggregations**, and **scans** across large tables benefit most from Photon's optimization.

Workloads that frequently access disk, process wide tables, or repeatedly transform data also see significant performance gains. For example, a data analyst running hourly aggregation queries across millions of rows will experience faster query completion times with Photon enabled.



Without Photon, the same workloads rely on standard Spark execution, which may suffice for simple transformations but lacks the performance optimizations Photon provides. Simple batch ETL jobs that process small datasets or complete in under two seconds typically see minimal benefit from Photon. In these cases, the overhead of enabling Photon may not justify the additional compute cost.

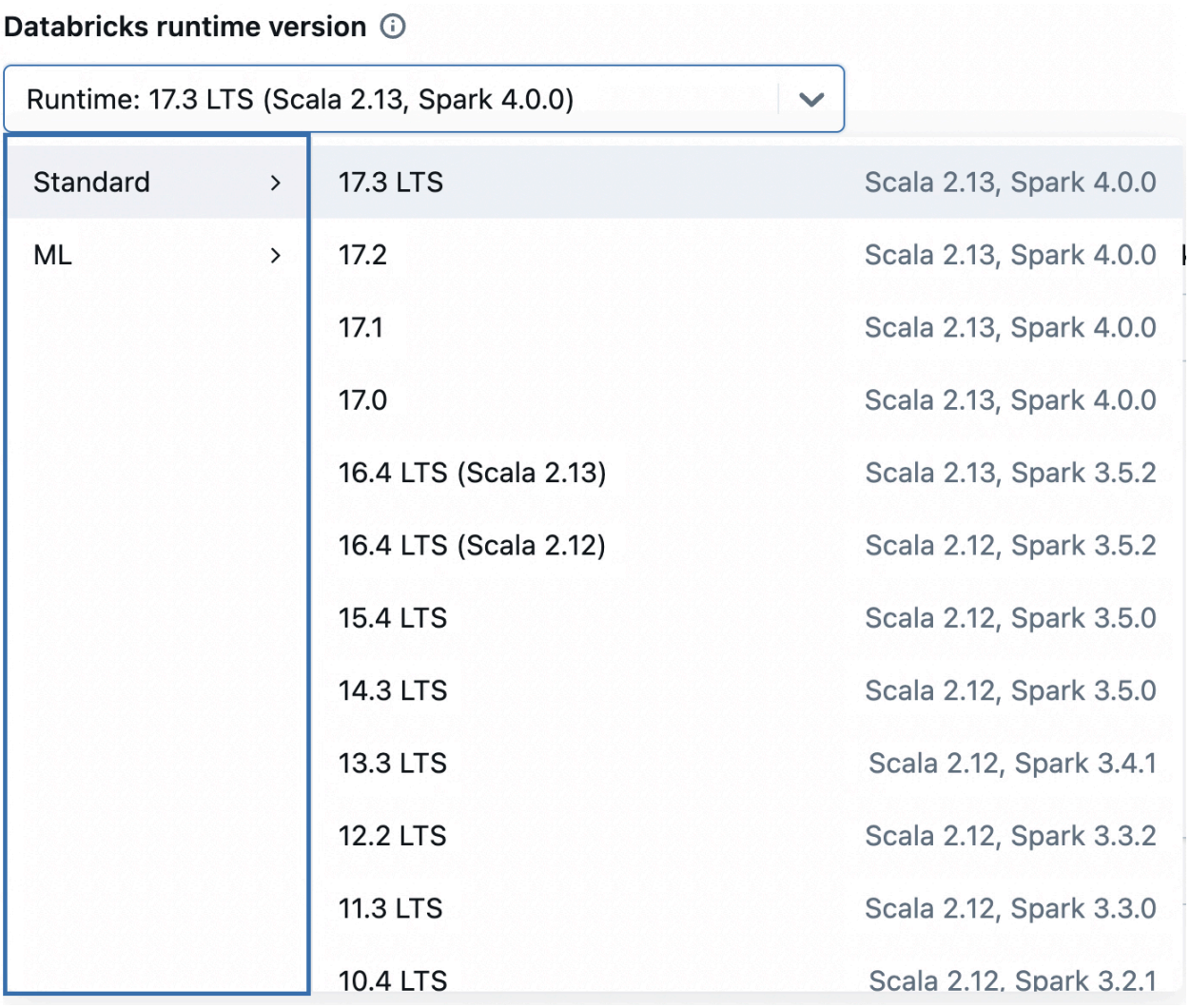
Photon is enabled by default on Databricks Runtime 9.1 LTS and above. You can verify or change this setting during compute creation under the Performance section. Keep in mind that **Photon isn't supported on GPU-enabled clusters**. If your workload requires GPU instances for machine learning or deep learning tasks, you must disable Photon.

Select Databricks Runtime and Spark version

Databricks Runtime provides the core components that run on your compute resource, including Apache Spark and additional optimizations. The runtime version you select determines which features are available and how your code interacts with preloaded packages.

Runtime selection impacts your workload in different ways depending on your use case. For **all-purpose compute** used in interactive development, the most current runtime version ensures you have the latest optimizations and compatibility with modern packages. This approach works well when you're actively developing notebooks and exploring data, as you benefit from continuous improvements.

For **job compute** running operational workloads, stability becomes more important than having the newest features. **Long Term Support (LTS)** runtime versions provide extended compatibility, allowing you to thoroughly test your workload before upgrading. A data engineering pipeline that runs daily transformations benefits from this stability, as unexpected runtime changes won't disrupt production processes.

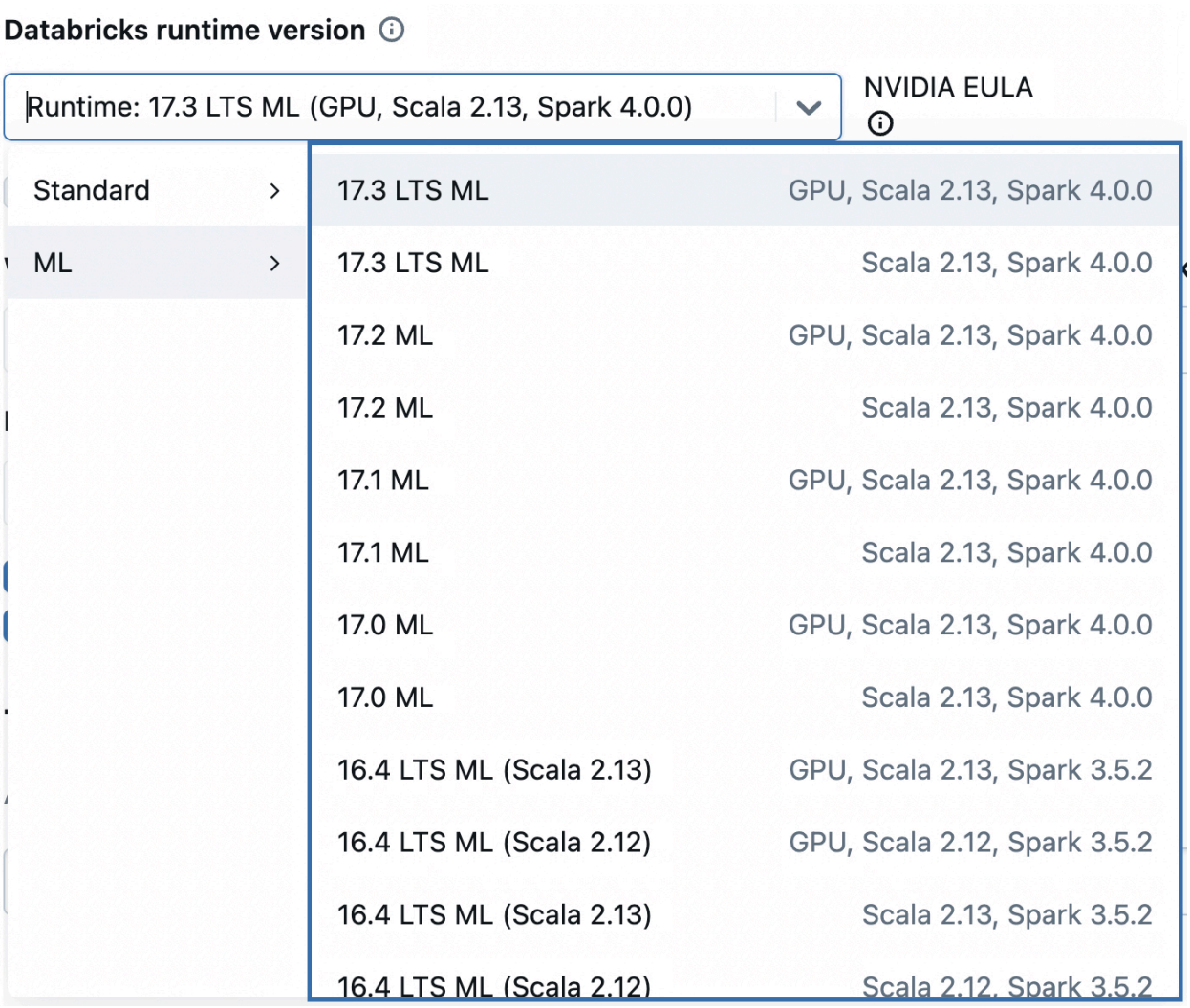


The **Spark version** is tied to your selected runtime, so choosing a runtime automatically determines which version of Spark you're using. Newer runtimes include more recent Spark versions with additional features and bug fixes. Older runtimes provide stability but may lack features introduced in recent Spark releases. When planning upgrades, consider testing your workload on the new runtime version in a development environment before applying changes to production compute resources.

Configure machine learning environments

Machine learning workloads require specialized runtime environments and hardware configurations. **Databricks Runtime ML** includes pre-installed machine learning libraries, GPU drivers, and frameworks like CUDA that support deep learning tasks.

When you configure compute for machine learning, start by selecting a **Databricks Runtime ML** version instead of the standard runtime. This runtime comes with popular libraries already installed, reducing setup time and ensuring compatibility between packages. For initial model experimentation, a **single-node** compute resource with a large instance type provides sufficient resources while minimizing shuffle overhead.



GPU instances enable training of deep learning models that would be impractical on CPU-only compute. Models involving image recognition, natural language processing, or neural networks benefit significantly from GPU acceleration. With GPU-enabled instances, operations that would take hours on CPUs complete in minutes. Keep in mind that **Photon must be disabled when using GPU instances**, as these features aren't compatible.

5. Install libraries for compute

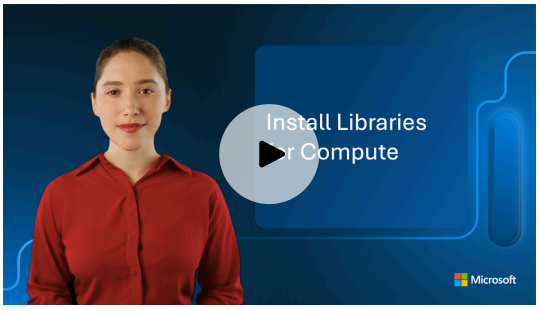
https://learn.microsoft.com/en-us/training/modules/select-and-configure-compute/5-install-libraries-for-compute

Install libraries for compute

Completed

When you run notebooks and jobs on Azure Databricks compute, you often need third-party packages or custom code that isn't included in the default runtime. Installing **libraries** at the **cluster level** ensures that every notebook and job using that compute has access to the same dependencies, creating a consistent execution environment.

Understanding how to install libraries effectively becomes critical as your data engineering workflows grow in complexity. You need to know which installation method to use, where to store library files, and how access modes affect your options.



Understand compute-scoped libraries

Compute-scoped libraries install on a cluster and become available to all notebooks and jobs that run on that cluster. Unlike **notebook-scoped libraries** that install only for a specific notebook session, compute-scoped libraries persist across cluster restarts and provide a shared environment for all users.

When you install a library at the cluster level, Azure Databricks automatically reinstalls it every time the cluster starts. This behavior ensures consistency—you don't need to manually reinstall dependencies after stopping and restarting a cluster. All notebooks attached to the cluster can import and use the installed packages immediately.

> **Note**
>
> To install libraries on a cluster, you must have **CAN MANAGE** permission on that cluster. This permission allows you to modify cluster configuration, including adding and removing libraries. Without this permission, you won't be able to access the library installation interface.

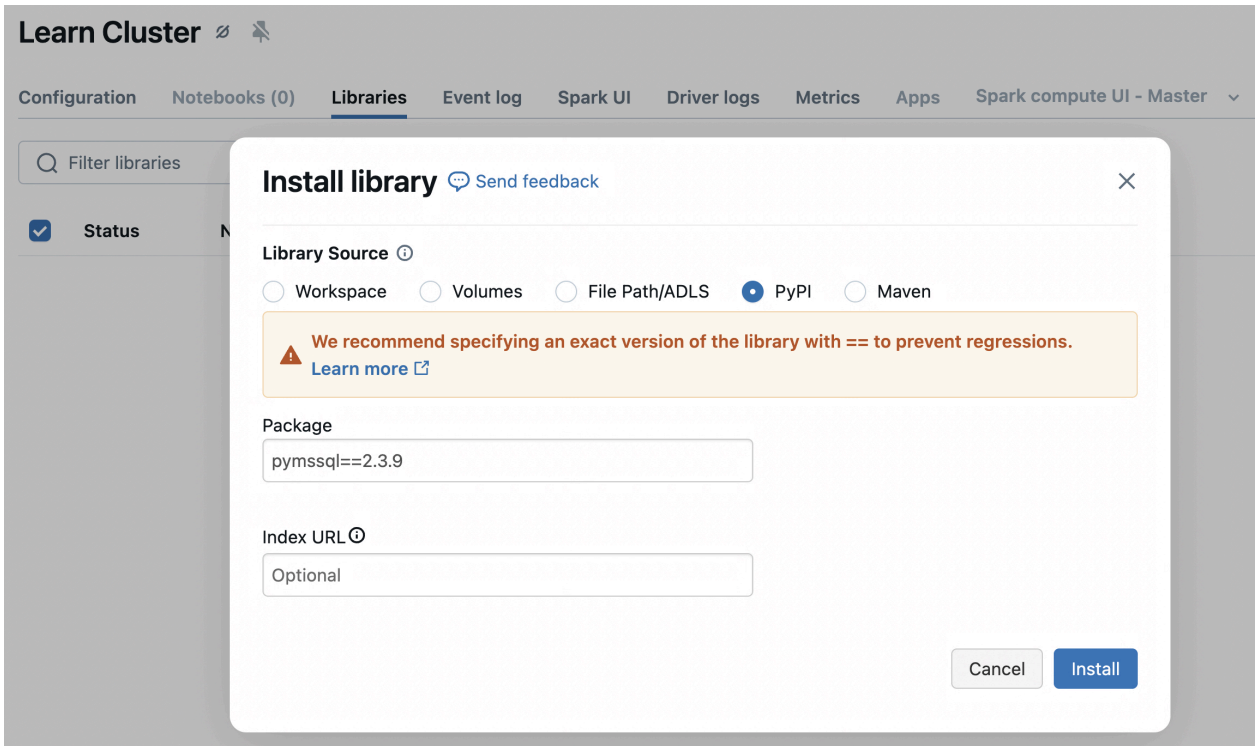
Compute-scoped libraries support **Python wheels**, **Java JAR files**, and **R packages**. You can install them from package repositories like **PyPI** and **Maven**, or from files stored in **workspace files**, **Unity Catalog volumes**, or cloud object storage. The installation method you choose depends on your library type, cluster access mode, and organizational security requirements.

However, compute-scoped libraries have an important limitation: any library you install affects every notebook on the cluster. If different teams need conflicting versions of the same library, you'll need separate clusters or notebook-scoped installations to avoid conflicts.

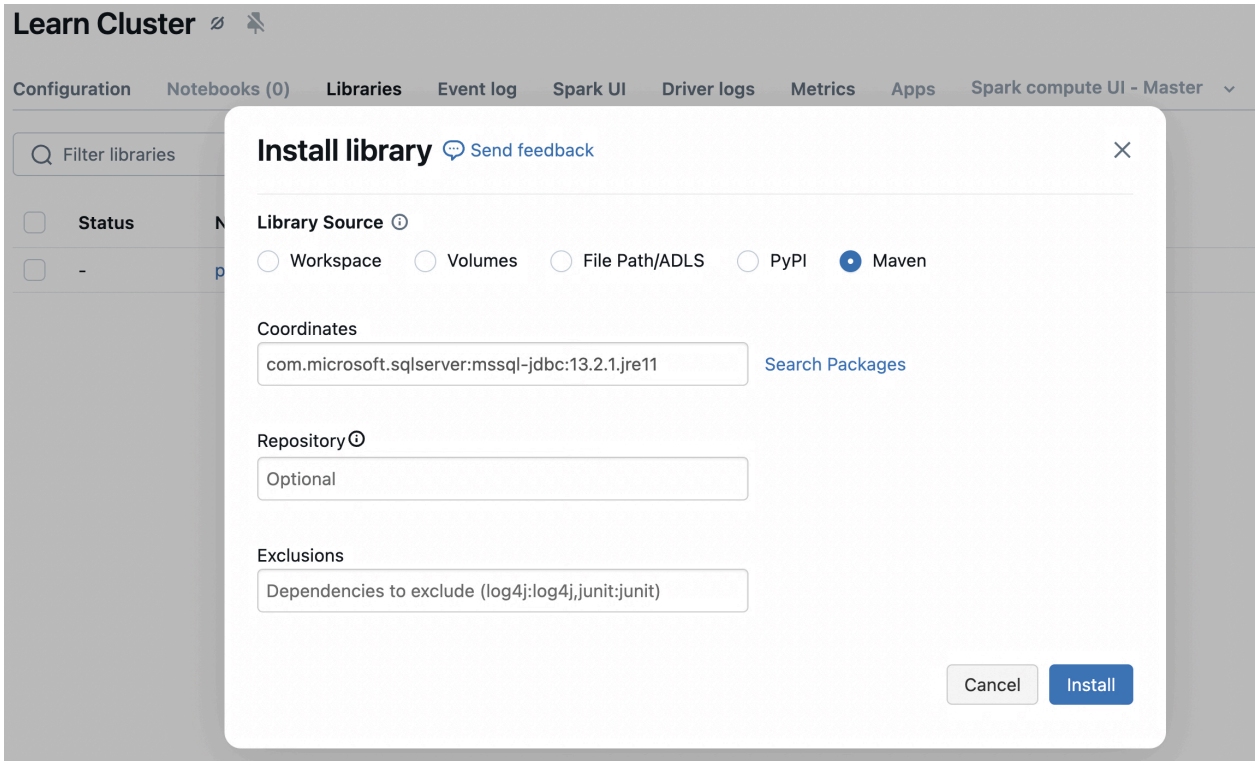
Install libraries from package repositories

Package repositories provide the most common way to install libraries. PyPI hosts Python packages, Maven hosts Java and Scala libraries, and **CRAN** hosts R packages. These repositories automatically handle **dependency resolution** and version management.

To install a library from PyPI, select **PyPI** as the library source and enter the package name. For production workloads, specify an exact version to ensure reproducibility: `pymssql==2.3.9`. Without a version number, Azure Databricks installs the latest available version, which might change between installations and potentially break your code.



Maven libraries require **coordinates** in the format `groupId:artifactId:version`. For example, to install the Microsoft JDBC Driver for SQL Server library, you would use `com.microsoft.sqlserver:mssql-jdbc:13.2.1.jre11`. You can search for packages directly in the installation dialog if you don't know the exact coordinates. Maven also supports excluding specific transitive dependencies that might conflict with other installed libraries.



For R packages from CRAN, provide the package name. Unlike Python and Java libraries, CRAN installations always pull the latest version from the configured mirror. To pin specific R package versions, you need to store the package files in workspace files or volumes instead of installing from CRAN.

With clusters configured in **standard access mode**, Maven coordinates and JAR file paths require **`allowlist` approval** before installation. This security measure ensures admins review and approve libraries that run on shared compute resources.

> **Note**
>
> To learn more about configuring and managing `allowlists` for libraries, see the documentation.

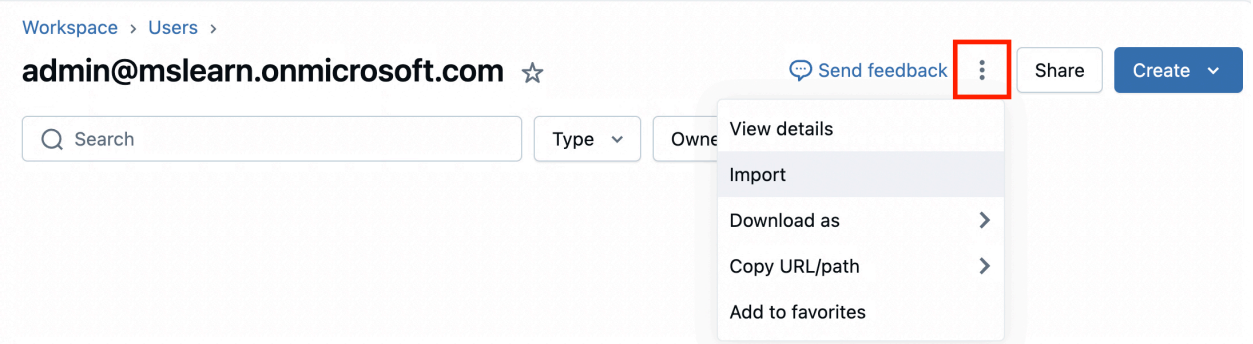
Install libraries from files

Storing library files in **workspace files** or **Unity Catalog volumes** gives you precise control over which library versions your clusters use. This approach works well when you need libraries not available in public repositories, custom packages you've built internally, or specific versions no longer available from package repositories.

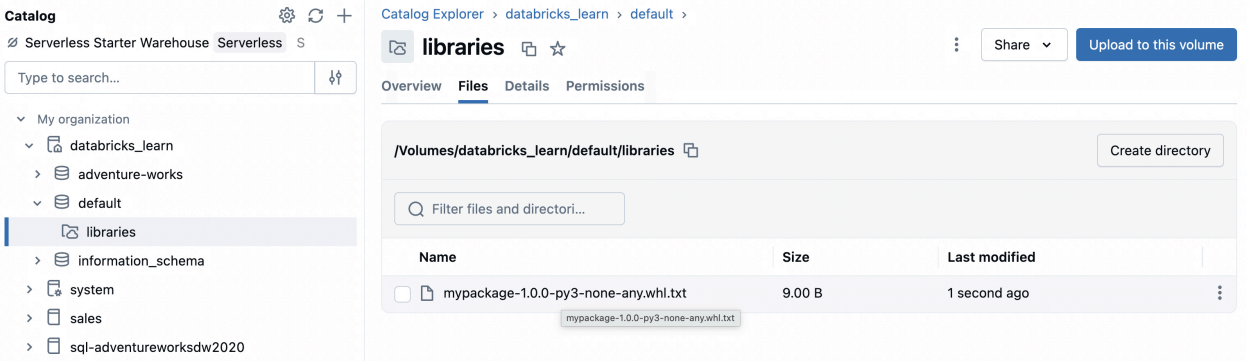
Using workspace files and Unity Catalog volumes for library installation maintains centralized management rather than bypassing security controls with ad-hoc installations like direct pip3 commands or unmanaged custom scripts executed from notebooks. Unity Catalog volumes provide

enhanced governance through Unity Catalog's access control model, ensuring that all library installations are tracked with audit logs and protected by fine-grained permissions.

Workspace files provide a convenient location for library storage with a 500 MB file size limit. To install a library from workspace files, upload your wheel, JAR, or requirements.txt file through the workspace Import dialog, then reference it during library installation using a path like `/Workspace/Users/you@example.com/libraries/mypackage-1.0.0-py3-none-any.whl`.



Unity Catalog volumes offer enhanced security and governance for library storage. You control access through Unity Catalog permissions, ensuring only authorized users can read or modify library files. Upload files to a volume through Catalog Explorer, then install them using a path like `/Volumes/main/engineering/libraries/mypackage-1.0.0-py3-none-any.whl`. The identity used for installation must have **READ VOLUME permission** on the specified volume.



Python **requirements.txt files** work with both workspace files and volumes in Databricks Runtime 15.0 and above. These files let you define multiple package dependencies in a single file, making it easier to maintain consistent environments across clusters. Upload the requirements.txt file and install it just like any other library—Azure Databricks automatically installs all listed packages.

For clusters with standard access mode, you must add library file paths to the `allowlist` before installation. This applies to both workspace files and volumes, ensuring admins approve the libraries used on shared compute.

Use init scripts for advanced configuration

Init scripts run shell commands during **cluster startup**, before the Spark driver and executors start. While Databricks **doesn't recommend** using init scripts for library installation—cluster-scoped

libraries provide a better approach—init scripts prove useful for system-level **configuration** that libraries can't handle.

You might use init scripts to install system packages with `apt-get`, configure environment variables, or set up monitoring agents. For example, an init script could install a specialized database driver that requires system libraries, then configure connection parameters through environment variables. The script runs every time the cluster starts, ensuring your configuration persists across restarts.

Store init scripts in Unity Catalog volumes for clusters running Databricks Runtime 13.3 LTS and above. Create a shell script file, upload it to a volume, then configure the cluster to run the script by specifying its path like `/Volumes/main/engineering/scripts/setup.sh`. For standard access mode, add the init script path to the `allowlist` before configuring the cluster.

Init scripts execute sequentially in the order you specify. If any script returns a non-zero exit code, the cluster fails to start. This failure protection prevents clusters from running with incomplete or incorrect configuration. You can troubleshoot failed init scripts by configuring cluster log delivery and examining the init script logs.

Consider init scripts as a last resort for configuration needs that cluster-scoped libraries and cluster policies can't address. Using cluster policies to set environment variables and Spark configurations often provides a simpler, more maintainable solution than init scripts.

Configure libraries for standard access mode

Clusters configured with **standard access mode** provide the strongest security and isolation in Azure Databricks. This mode requires explicit approval for libraries and init scripts to prevent unauthorized code execution on shared compute resources.

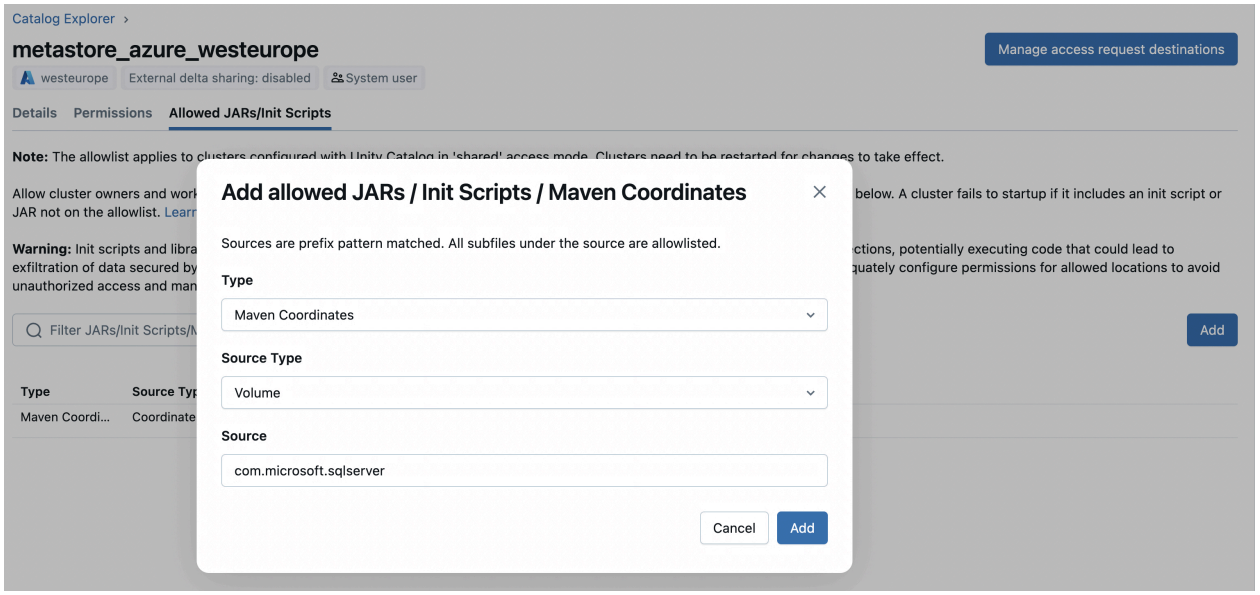
Before installing Maven libraries or JAR files on standard access mode clusters, a **metastore admin** must add them to the `allowlist`. Maven coordinates go on the `allowlist` using the format `groupId:artifactId:version`. You can `allowlist` all versions of a library with `groupId:artifactId`, or all artifacts in a group with just `groupId`. For JAR files stored in volumes or object storage, `allowlist` the file path or directory path.

Init scripts require separate `allowlist` entries even if stored in the same location as JAR files. When allow listing a path, Azure Databricks uses prefix matching—adding `/Volumes/prod-libraries/` to the `allowlist` permits all files and subdirectories within that location. Include a trailing slash to prevent unintended prefix matches at the directory level.

The `allowlist` only grants permission to use a path for library or init script installation. You still need appropriate data access permissions. For volumes, the installer identity must have `READ VOLUME` permission. For standard access mode, the cluster owner's identity validates these permissions during library installation.

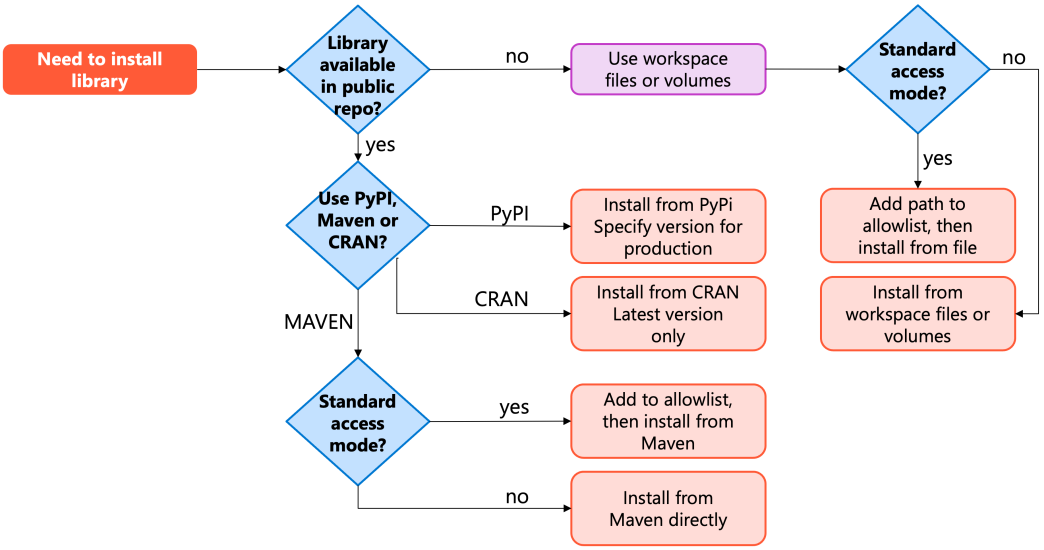
To configure the `allowlist`, metastore admins use Catalog Explorer, selecting the metastore settings and navigating to the **Allowed JARs/Init Scripts** section. This centralized control ensures

that security teams can review and approve all libraries used across the organization's compute resources, maintaining governance without blocking productivity.



Choose the right installation method

Different library installation methods suit different scenarios. The following diagram illustrates a decision flow to help you select the appropriate installation approach:



6. Configure compute access

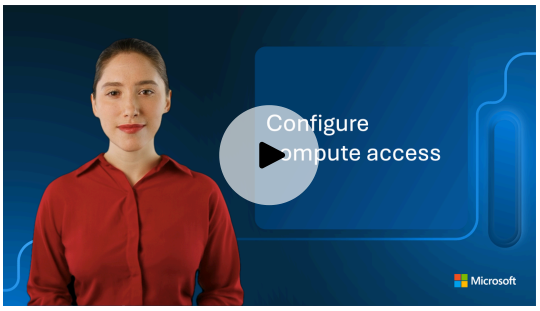
https://learn.microsoft.com/en-us/training/modules/select-and-configure-compute/6-configure-compute-access-perms

Configure compute access

Completed

Managing access to compute resources protects sensitive data while enabling collaboration across your data engineering teams. You control who can attach notebooks, restart clusters, or modify compute configurations through **permission levels** assigned in the Azure Databricks workspace.

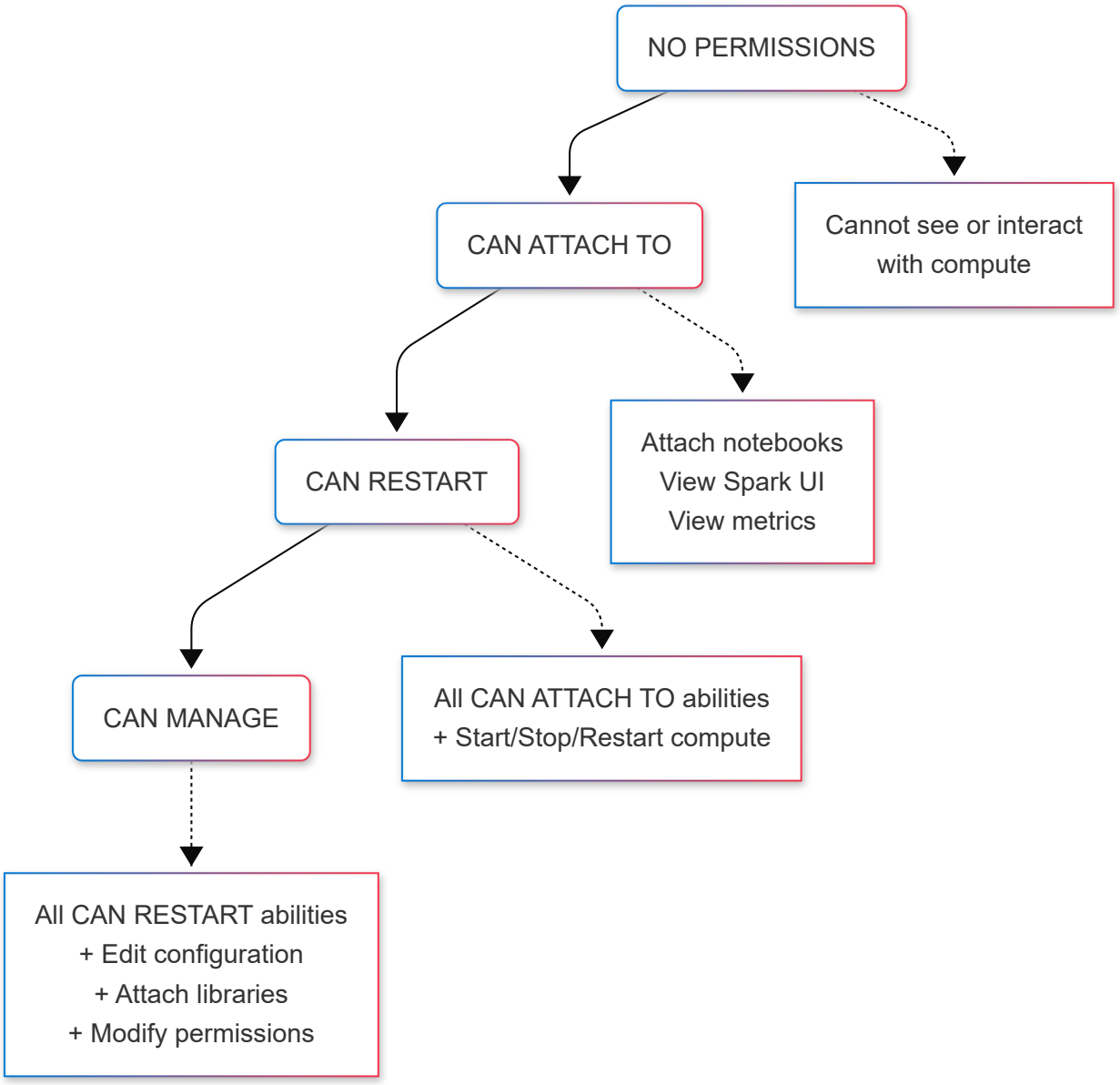
Proper access configuration prevents unauthorized users from viewing driver logs containing secrets, controls who can consume compute resources, and ensures teams work efficiently without conflicting infrastructure changes. With **Unity Catalog** enabled, you can assign compute resources to entire groups with automatic permission scoping.



Understand compute permission levels

Azure Databricks offers **four permission levels** for compute resources, each granting progressively more capabilities. These permissions operate within your workspace and are distinct from Azure subscription-level access controls.

The following diagram shows the hierarchy of compute permissions and the capabilities each level provides:



NO PERMISSIONS prevents any interaction with the compute resource. Users can't see the compute in their workspace, attach notebooks to it, or view metrics or logs. This default state ensures compute resources remain private until you explicitly grant access.

CAN ATTACH TO enables basic compute usage. Users with this permission can attach their notebooks to the compute, view the Spark UI to monitor job execution, and access compute metrics for performance analysis. However, they can't control the compute lifecycle—no starting, stopping, or restarting. This permission level works well for analysts who need to run queries but shouldn't manage infrastructure.

With this permission level, users on compute configured with **No isolation shared access mode** can view service account keys in log4j files. Because of this security consideration, No isolation shared access mode is legacy and should be avoided. **Standard** or **Dedicated** access modes provide better isolation.

CAN RESTART includes everything from `CAN ATTACH TO` plus lifecycle management capabilities. Users can terminate, start, and restart the compute resource. This permission suits team leads or senior engineers who need to manage compute availability without full administrative control. You might grant `CAN RESTART` to someone who needs to restart a misbehaving cluster but shouldn't change its configuration.

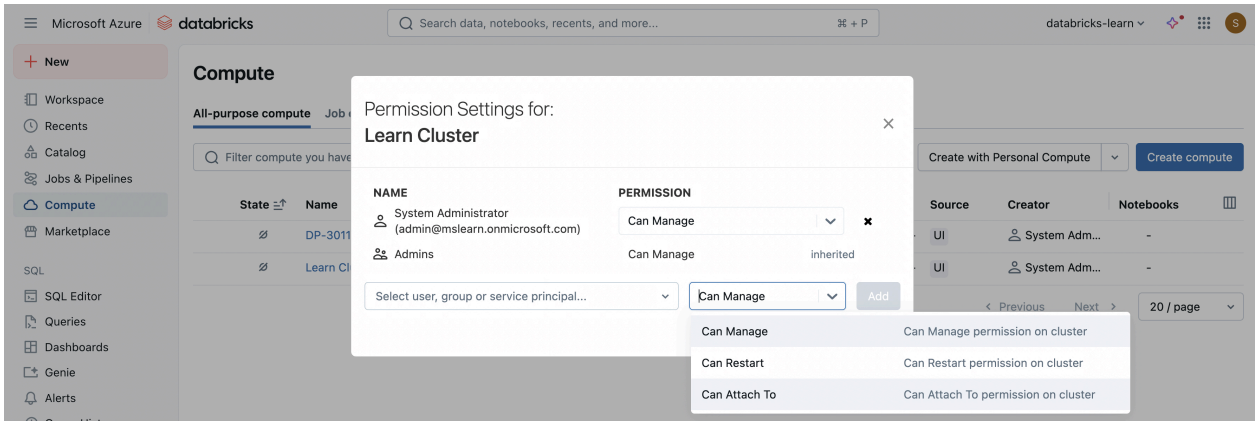
CAN MANAGE provides complete control over the compute resource. Beyond all `CAN RESTART` capabilities, users can edit compute configuration, attach libraries, resize the cluster, and modify permissions for other users. Workspace admins automatically receive `CAN MANAGE` permission on all compute resources. Users who create a compute resource become its owner with `CAN MANAGE` rights.

By default, only users with `CAN MANAGE` permission can view **driver logs** on job clusters, dedicated access mode clusters, and standard access mode clusters. This restriction exists because driver logs expose secrets through stdout and stderr streams. You can adjust this behavior using the Spark configuration property `spark.databricks.acl.needAdminPermissionToViewLogs`, though changing it may create security risks.

Configure permissions in the workspace

You set compute permissions through the Azure Databricks workspace UI, not the Azure portal. This workspace-level configuration operates independently from Azure subscription permissions, though users still need workspace access to benefit from compute permissions.

To configure permissions on a compute resource, select **Compute** from the workspace sidebar. Locate the compute resource you want to manage and select the **Permissions** option from its actions menu. The permissions dialog shows current access grants and allows you to add or modify permissions.



Select **Add** to grant new permissions. Search for **users** or **groups** by name or email address. Choose the appropriate permission level from the dropdown menu— `CAN ATTACH TO`, `CAN RESTART`, or `CAN MANAGE`. Multiple users and groups can have different permission levels on the same compute resource, giving you flexibility in access control.

When you modify permissions, changes take effect immediately. Users gain or lose access without requiring a compute restart. This immediate application means you can quickly respond to team changes, onboard new members, or revoke access when someone leaves a project.

Consider granting permissions to **groups** rather than individual users when possible. Group-based permissions simplify management as team membership changes. When someone joins the data engineering team, adding them to the appropriate group automatically grants the correct compute access. Similarly, removing someone from the group revokes their permissions without manual cleanup.

Manage workspace-level entitlements

Beyond individual compute permissions, Azure Databricks provides **workspace-level entitlements** that control what users can do across the entire workspace. These entitlements differ from compute permissions—they govern user capabilities rather than access to specific resources.

The **unrestricted cluster creation** entitlement allows non-admin users to create clusters without size restrictions. By default, only workspace administrators can create clusters of any size. When you enable this entitlement for a user or group, they gain the ability to provision clusters with any number of worker nodes and any VM configuration, but they still don't receive workspace admin privileges.

This entitlement follows the principle of least privilege by letting you grant cluster creation capabilities without elevating users to full workspace administrators. Users with this entitlement can create compute resources that match their workload requirements but can't perform other administrative functions like managing workspace settings, viewing all users' notebooks, or modifying account-level configurations.

To grant the unrestricted cluster creation entitlement:

1. In your workspace, select **Settings** from the sidebar.
2. Navigate to **Identity and access** > **Users** or **Groups**.
3. Locate the user or group you want to grant the entitlement to.
4. Select the user or group, then find the **Entitlements** section.
5. Enable the **Allow cluster creation** entitlement.

When designing your access control strategy, consider which users genuinely need unrestricted cluster creation. Data engineers building production pipelines often require this capability, while data analysts might only need access to pre-configured shared clusters. Grant this entitlement to roles and teams rather than individuals when possible to simplify management as your organization grows.

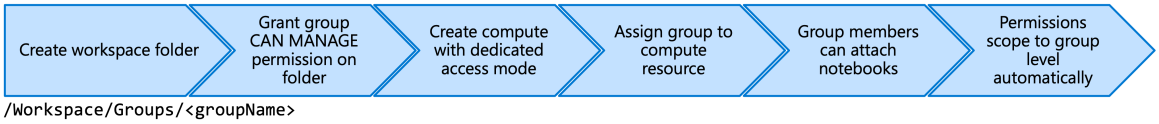
Dedicated group access mode

Standard access mode allows multiple users to share compute resources securely, with Lakeguard providing isolation between user workloads. **Dedicated access mode** takes a different approach by

assigning the entire compute resource to a single user or group, with user permissions automatically scoping down to match the assigned entity's permissions.

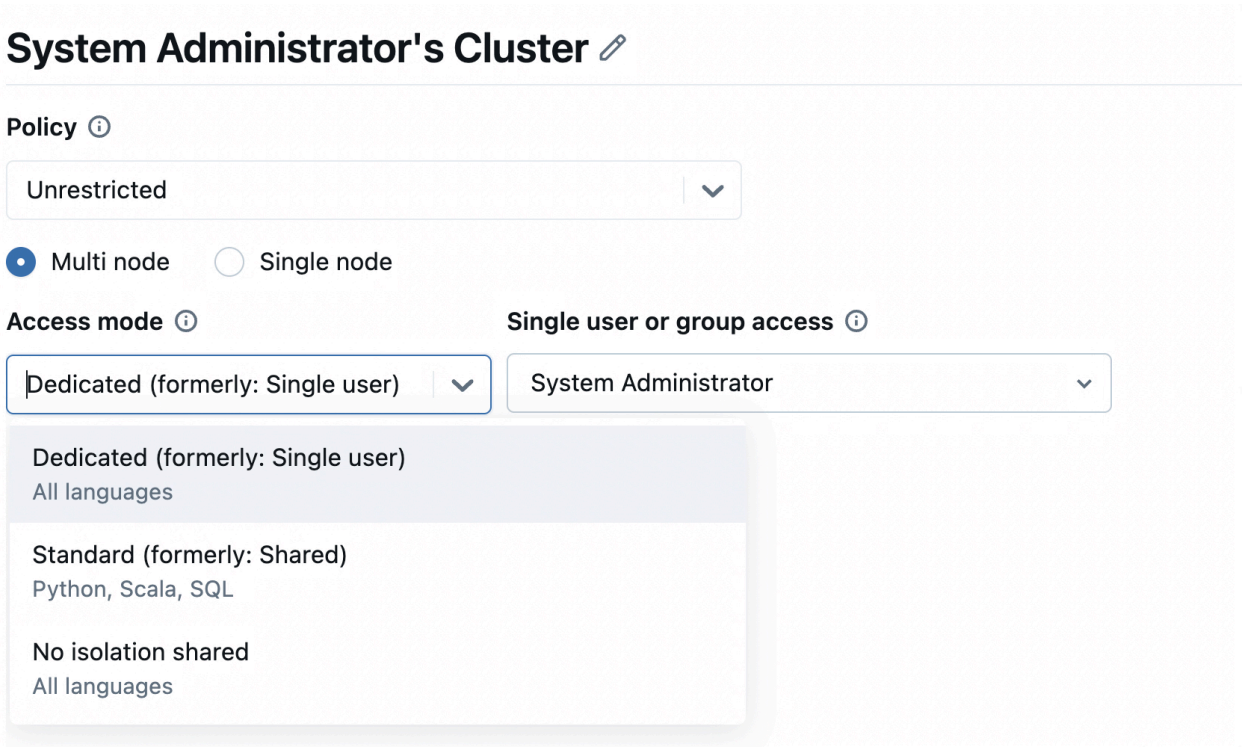
This **permission scoping** enables secure group collaboration on workloads that standard access mode doesn't support, including **Databricks Runtime for ML**, **RDD APIs**, and **R language** execution. When a user connects to a group cluster, their individual permissions temporarily reduce to only what the group possesses, preventing privilege escalation while maintaining necessary functionality.

The following diagram illustrates how to set up dedicated group access:



To create a compute resource dedicated to a group, you must enable Unity Catalog on your workspace and use Databricks Runtime 15.4 or above. The assigned group needs `CAN MANAGE` permission on a workspace folder where members can store notebooks, MLflow experiments, and other workspace artifacts used on the group cluster.

When creating the compute, expand the **Advanced** section and select **Dedicated (formerly: Single-user)** under **Access mode**. In the **Single user or group** field, choose the group that should own this resource. Only members of the selected group can attach notebooks to this compute resource.



For effective group cluster management, create a dedicated **workspace folder** at `/Workspace/Groups/<groupName>` for each group using this pattern. Grant the group `CAN MANAGE`

permission on this folder. All notebooks, experiments, and workspace assets the group uses should reside in this folder to avoid permission errors.

Group clusters introduce specific behavior changes you should understand. When you create data objects using the group cluster, the group becomes the object's owner, not the individual user. For example, if you run `CREATE SCHEMA human_resources` on a group cluster, the group owns that schema. Individual user permissions aren't enforced because all group members access shared Spark APIs and the same compute environment.

Best practices for compute access

Apply the **principle of least privilege** when granting compute permissions. Start with the minimum permission level users need to accomplish their work. Grant `CAN ATTACH TO` for analysts running queries, `CAN RESTART` for engineers managing cluster availability, and reserve `CAN MANAGE` for administrators and resource owners.

Use **groups** instead of individual user assignments whenever possible. Create groups that mirror your organizational structure—data engineering teams, analytics teams, or project-specific groups. Grant permissions to these groups, then manage membership through the groups themselves. This approach scales better as your organization grows and reduces the likelihood of orphaned permissions when people change roles.

For production workloads, create **dedicated compute resources with restricted access**. Production clusters shouldn't be widely accessible to development users. Limit `CAN MANAGE` permissions to the specific team responsible for that workload, and grant `CAN ATTACH TO` only when users have legitimate business needs to run production queries.

Pay attention to **driver log access**, especially when debugging applications that use secrets. The default configuration restricts log viewing to users with `CAN MANAGE` permissions, protecting secrets from exposure. If you must grant broader log access by setting `spark.databricks.acl.needAdminPermissionToViewLogs` to `false`, ensure secrets are managed through Databricks secret scopes with appropriate access controls rather than being hardcoded.

When using **group clusters**, establish clear folder structures and permission patterns before deployment. Create the group workspace folders, assign appropriate permissions, and document the expected usage patterns. Ensure MLflow tracking URIs point to group folders and that AutoML runs specify group-accessible experiment directories. This upfront organization prevents permission errors and simplifies troubleshooting.

Monitor compute access through **audit logs** regularly. Review who has `CAN MANAGE` permissions on critical resources, verify that former team members no longer have access, and ensure permission grants align with current organizational needs. The `system.access.audit` table provides detailed access history for compliance and security reviews.

Avoid using No isolation shared access mode entirely. This legacy mode exposes service account keys to users with `CAN ATTACH TO` permissions and lacks the security guarantees of standard or

dedicated access modes. If you encounter existing No isolation shared clusters, migrate them to standard or dedicated access mode to improve security posture.

7. Exercise - Select and Configure Compute in Azure Databricks

https://learn.microsoft.com/en-us/training/modules/select-and-configure-compute/7-exercise-select-configure-compute

Exercise - Select and Configure Compute in Azure Databricks

Completed

Now it's your chance to select and configure compute in Azure Databricks. In this lab, you create and configure an all-purpose cluster in Azure Databricks, install libraries both cluster-scoped and notebook-scoped, and use the faker library to generate and analyze synthetic patient admission records with PySpark.

> **Note**
>
> To complete this lab, you need an Azure subscription in which you have administrative access.

Launch the exercise and follow the instructions.

Launch Exercise

8. Module assessment

https://learn.microsoft.com/en-us/training/modules/select-and-configure-compute/8-knowledge-check

Module assessment

Completed

9. Summary

https://learn.microsoft.com/en-us/training/modules/select-and-configure-compute/9-summary

Summary

Completed

Selecting and configuring compute resources effectively determines both the success and cost of your Azure Databricks workloads. Throughout this module, you explored the compute options available and learned how to match them to specific scenarios. Serverless compute provides the fastest path to productivity with minimal overhead, while classic compute offers complete control when you need specialized features. SQL warehouses optimize analytical queries, and job clusters ensure automated workflows run efficiently without idle costs.

Performance configuration extends beyond choosing a compute type. You discovered how node types, autoscaling settings, and termination policies balance cost with responsiveness. Memory-optimized instances handle large joins more efficiently, while compute-optimized instances excel at CPU-intensive transformations. Photon acceleration doubles query performance for SQL workloads, and instance pools reduce startup time when justified by usage patterns.

Access management and library installation complete the compute configuration picture. Permission levels enable the principle of least privilege, granting users exactly the access they need without unnecessary risk. Dedicated group access modes bring secure collaboration to workloads requiring RDD APIs or R language. Library installation from repositories, files, or volumes ensures your code has the dependencies it needs while maintaining security through allowlist controls.

As you implement these patterns in your organization, start with serverless options for new workloads, then move to classic compute only when specific limitations require it. Monitor actual usage to optimize configurations over time, apply permission controls thoughtfully, and maintain consistent library management practices across teams. The compute decisions you make today shape the performance, security, and cost efficiency of your data platform for months to come.

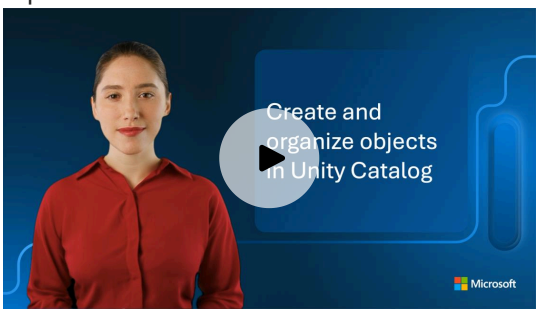
Create and organize objects in Unity Catalog

1. Introduction

https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/1-introduction

Introduction

Completed



Your organization needs to manage data across development, testing, and production environments while enforcing clear security boundaries. Different teams require access to different datasets, and business users need a way to discover and query data without writing complex SQL. **Unity Catalog** provides a **three-layer namespace**—catalogs, schemas, and tables—that lets you organize data assets with precision while maintaining centralized governance.

Creating and organizing Unity Catalog objects establishes the foundation for your data platform. You start by designing **catalogs** that separate environments or business domains, then organize **schemas** within those catalogs to group related datasets. Within schemas, you create **tables** for structured data and **volumes** for files, applying consistent naming conventions that make your data discoverable. Advanced features like **foreign catalogs** connect external databases, while **AI/BI Genie instructions** help business users interact with your data using natural language.

By the end of this module, you'll be able to design and implement a comprehensive Unity Catalog structure that scales with your organization's needs while maintaining clear governance boundaries.

2. Apply naming conventions

https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/2-apply-naming-conventions

Apply naming conventions

Completed

Naming conventions are fundamental to effective data governance in Azure Databricks Unity Catalog. When you apply consistent naming patterns across catalogs, schemas, and tables, you enable your team to quickly understand data organization, enforce access controls, and maintain compliance with organizational standards.

Understand Unity Catalog naming requirements

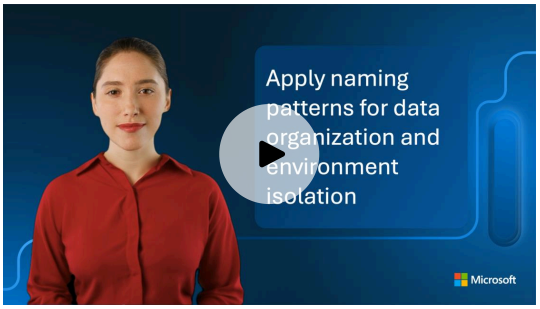
Unity Catalog imposes specific technical constraints on object names that you must follow when designing your naming conventions. These constraints ensure compatibility across the platform while supporting international characters and diverse organizational needs.

All Unity Catalog object names are limited to 255 characters and are stored in lowercase, regardless of how you create them. This means `SalesData`, `salesdata`, and `SALESDATA` all resolve to the same object. When you reference objects in queries, Unity Catalog performs case-insensitive matching, simplifying query syntax while maintaining consistency.

Certain characters are prohibited in object names. You can't use periods (`.`), spaces, forward slashes (`/`), or ASCII control characters. However, you can include hyphens and other special characters by enclosing the name in backticks. For example, `` `sales-data` `` is valid, while `sales.data` isn't because periods serve as namespace separators in the three-level hierarchy.

Column names follow different rules. Unity Catalog preserves the casing you specify for column names, though queries remain case-insensitive. You can use special characters in column names by enclosing them in backticks, giving you flexibility for descriptive field names while maintaining query simplicity.

Apply naming patterns for data organization



Effective naming patterns balance descriptiveness with brevity, helping your team understand data purpose without creating unwieldy identifiers. The three-level namespace structure of Unity Catalog — `catalog.schema.table` —provides natural boundaries for applying meaningful conventions.

```
sales_data (Catalog)
└── bronze (Schema - raw data)
    ├── raw_orders (Table)
    └── raw_customers (Table)

└── silver (Schema - cleaned / validated)
    ├── cleaned_orders (Table)
    └── validated_customers (Table)

└── gold (Schema - aggregated / business-level)
    ├── sales_summary (Table)
    ├── customer_lifetime_value (Table)
    └── vw_customer_stats (View example: prefix views with vw_)
```

Use **lowercase with underscores** as your primary pattern. Names like `customer_data`, `sales_summary`, and `product_inventory` are immediately readable and comply with Unity Catalog requirements. Avoid camelCase or PascalCase, as Unity Catalog converts all names to lowercase anyway, potentially creating confusion between what you specify and what the system stores.

Choose **descriptive but concise names** that clearly indicate content without excessive detail. `sales_summary_gold` effectively communicates both the domain and data quality level. In contrast, `sales_final_report_v1_updated` includes unnecessary version information that belongs in metadata or table properties rather than the name itself.

Align schema names with data layers when implementing a medallion architecture. Use **bronze** for raw ingested data, **silver** for cleaned and validated data, and **gold** for aggregated business-level data. This pattern makes the data transformation pipeline immediately apparent. For example, `sales_data.bronze.raw_orders` clearly indicates raw order data, while `sales_data.gold.monthly_revenue` represents a refined business metric.

Prefix specialized objects to distinguish them from base tables. Add `vw_` for views and materialized views (`vw_customer_stats`), and `tmp_` for temporary staging tables (`tmp_import_staging`). This convention helps you identify object types at a glance when browsing the catalog or writing queries.

Consider your **catalog naming strategy** based on whether you're organizing by business domain or environment. Domain-based catalogs like `sales_data`, `marketing_data`, and `finance_data` provide clear functional boundaries and align with data ownership models. Environment-based catalogs like `dev`, `test`, and `prod` separate workspaces by lifecycle stage, reducing risk of accidental production changes.

Design naming strategies for environment isolation

Environment isolation is critical for preventing accidental data modifications and maintaining clear separation between development, testing, and production workloads. Your naming strategy should make environment boundaries immediately obvious to anyone working with the data.

```
unity_catalog
└─ catalogs
   ├─ sales_dev
   │  ├─ raw               (schema)
   │  ├─ staging           (schema)
   │  └─ curated           (schema)
   │     ├─ orders_dev         (table)
   │     └─ customers_dev      (table)
   │
   ├─ sales_test
   │  ├─ raw
   │  ├─ staging
   │  └─ curated
   │     └─ orders_test
   │
   └─ sales_prod
      ├─ raw
      ├─ staging
      └─ curated
         ├─ orders
         └─ customers
```

Use separate catalogs for each environment when strong isolation is your priority. Catalogs named `sales_dev`, `sales_test`, and `sales_prod` provide complete separation at the highest level of the namespace. With this approach, you grant permissions at the catalog level, making access control straightforward. Developers work exclusively in the dev catalog, while production queries reference only the `prod` catalog, eliminating cross-environment contamination risk.

Combine domain and environment in catalog names when you need both dimensions. A pattern like `{domain}_{environment}` produces catalogs such as `sales_prod`, `sales_dev`, `marketing_prod`, and `marketing_dev`. This strategy scales well across multiple business domains while maintaining environment separation. However, it increases the total number of catalogs, so you need robust catalog management practices.

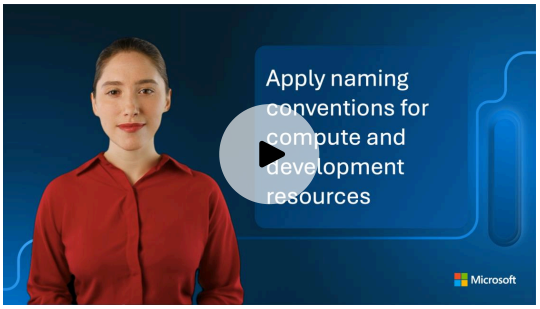
Organize by layer within environment catalogs when environment separation happens at the workspace level rather than the catalog level. Within a `prod` catalog, you might have schemas named `sales_bronze`, `sales_silver`, and `sales_gold`, while a separate `dev` catalog contains

`sales_bronze`, `sales_silver`, and `sales_gold` for development work. This approach consolidates all production data under a single catalog, simplifying Unity Catalog governance while maintaining clear layer boundaries.

For **external sharing scenarios**, choose catalog names that are meaningful to external partners without exposing internal organizational structure. A catalog named `customer_analytics_shared` clearly indicates both purpose and sharing status, while `internal_sales_raw_data` signals data not intended for external access. This convention helps prevent accidental sharing of sensitive data.

External storage paths should mirror your naming conventions for consistency. Following a pattern like `abfss://container@storage.dfs.core.windows.net/env/layer/domain/table/` aligns storage organization with Unity Catalog structure. For example, `abfss://datalake@company.dfs.core.windows.net/prod/gold/sales/monthly_revenue/` makes the relationship between storage and catalog objects transparent.

Apply naming conventions for compute and development resources



Beyond data objects, consistent naming conventions for compute resources, development artifacts, and operational components help you navigate the Azure Databricks workspace efficiently and maintain clear ownership boundaries.

Clusters

Name clusters according to their purpose and environment to make resource allocation transparent. Use patterns like `{environment}_{domain}_cluster` to produce names such as `dev_sales_cluster`, `prod_etl_cluster`, and `ad_hoc_analysis_cluster`. This convention immediately identifies whether a cluster is for development experimentation or production workloads, helping you manage costs and apply appropriate access controls.

Jobs and pipelines

Structure job names using the pattern `job_{layer}_{purpose}` to align with your data transformation pipeline. Examples include `job_bronze_orders_ingestion`, `job_silver_orders_transformation`, and `job_gold_sales_aggregation`. This naming pattern makes dependencies between jobs immediately visible and helps you trace data lineage across the medallion architecture.

For Lakeflow Spark Declarative Pipelines pipelines, use the prefix `pipe_` followed by the data domain or purpose: `pipe_orders_processing`, `pipe_customer_data_cleaning`.

Name streaming pipelines to include both source and target, following patterns like `stream_{source}_to_{target}`. Examples such as `stream_kafka_to_bronze` and `stream_iot_sensor_data` make data flow explicit without requiring pipeline documentation. This convention is especially valuable when managing multiple concurrent streaming workloads.

Notebooks and repositories

Organize notebooks in folder hierarchies aligned with your project structure and data layers. A recommended pattern mirrors the medallion architecture:

```
/Repos/
  sales_project/
    bronze/
      ingestion_orders/
    silver/
      transformation_orders/
    gold/
      sales_aggregation/
```

Individual notebook names should reflect their purpose and layer using patterns like `notebook_{layer}_{purpose}`. Examples include `notebook_bronze_orders_ingestion`, `notebook_silver_orders_transformation`, and `notebook_gold_sales_reporting`. This creates consistency between notebook names and the job names that execute them.

Repository names should identify the project or domain they contain. Use patterns like `repo_{project_name}` to produce names such as `repo_sales_project` and `repo_customer_analytics`. When you integrate with Git, this naming convention helps map Databricks repositories to their corresponding remote repositories.

Dashboards and monitoring

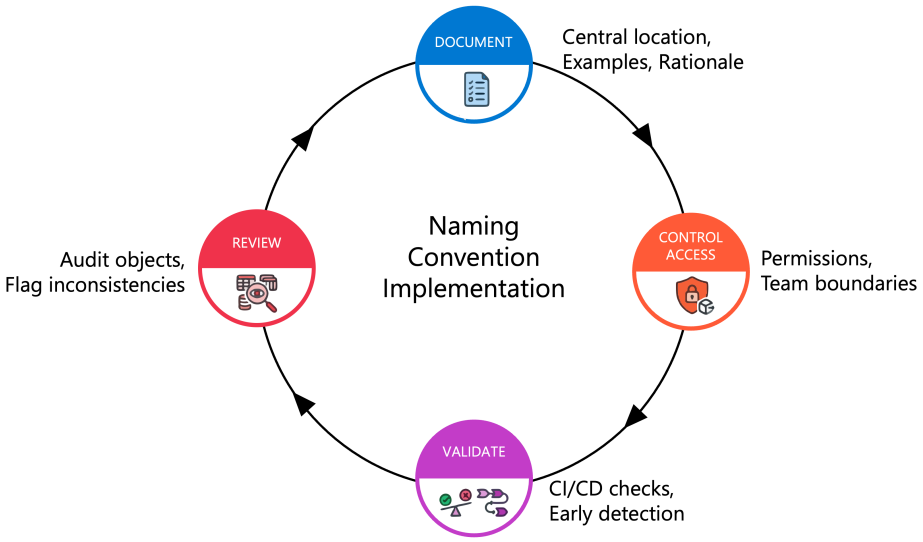
Name SQL dashboards to reflect their business purpose using patterns like `dashboard_{domain}_{purpose}`. Examples include `dashboard_sales_performance` and `dashboard_customer_trends`. This convention helps business users quickly locate relevant dashboards and distinguishes operational dashboards from exploratory analyses.

Individual SQL queries should be named descriptively using patterns like `query_{purpose}`. Names such as `query_monthly_sales_report` and `query_customer_retention_analysis` make query libraries searchable and help teams identify reusable queries for similar reporting needs.

Implement conventions across your organization

Successful implementation of naming conventions requires more than technical patterns—you need organizational alignment, documentation, and enforcement mechanisms that make conventions easy

to follow and hard to bypass.



Start by **documenting your conventions** in a central location accessible to all data team members. Include specific examples, decision criteria, and rationale for each pattern. A documented standard like "Use `{domain}_{layer}` for schema names, where domain is the business area and layer is bronze/silver/gold" removes ambiguity. Provide examples of both correct and incorrect names to illustrate the pattern clearly.

Control access through Unity Catalog permissions to limit who can create objects in specific catalogs and schemas. While you can't enforce naming patterns through permissions alone, restricting `CREATE TABLE` and `CREATE SCHEMA` privileges to specific teams helps maintain organizational boundaries. For example, grant the marketing team permissions only to the marketing catalog, reducing the risk of objects being created in the wrong location.

Validate names during CI/CD pipelines when deploying data assets through automation. Include checks in your deployment scripts that reject names violating your standards before objects are created. This automated validation catches errors early and maintains consistency across all environments.

Review existing objects regularly to identify naming inconsistencies and plan remediation. Use Unity Catalog's information schema to query object names and flag those that don't match your patterns. For newly identified legacy objects, decide whether to rename them for consistency or document them as exceptions with justification.

Balance flexibility with standardization based on your organization's needs. Strict enforcement works well for highly regulated environments where consistency is critical for compliance. More flexible guidelines may be appropriate for smaller teams or during periods of rapid experimentation, as long as you eventually converge on standards for production data.

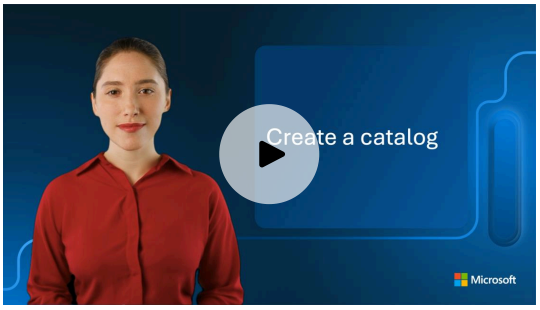
3. Create catalog

https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/3-create-catalog

Create catalog

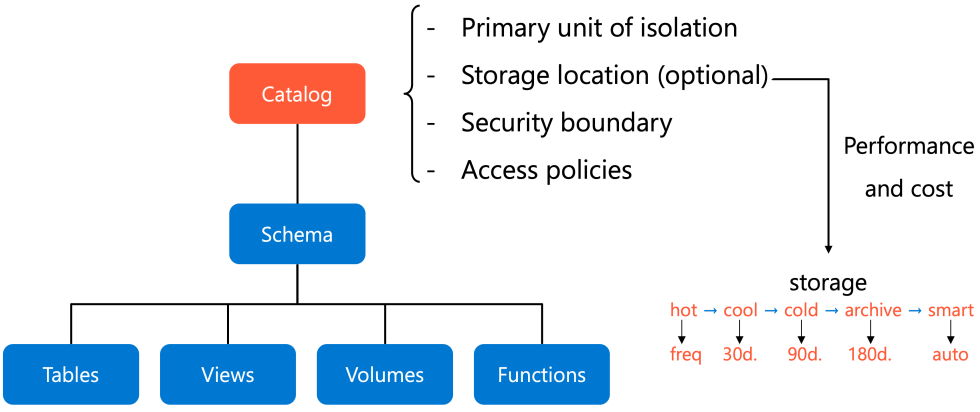
Completed

Your organization's data governance policies require clear separation between **development**, **testing**, and **production** environments. You need production data in dedicated storage, development work isolated from live systems, and sensitive customer information protected from operational data. **Unity Catalog's catalog structure** helps you meet these requirements while maintaining centralized governance and access control.



What is a catalog?

A **catalog** is the **top-level container** in Unity Catalog's **three-layer namespace**. Think of it as the foundation of your data organization, like building a house before decorating the rooms inside. Every data asset in Unity Catalog exists within this hierarchy: `catalog.schema.table`.



Catalogs serve as the primary unit of **data isolation** and organization. Each catalog can have its own **storage location**, **security boundaries**, and **access policies**. This physical separation ensures that your development experiments don't accidentally affect production data, and that sensitive information remains isolated from general-purpose datasets.

With only three layers available for organizing your data, understanding each layer's role becomes essential. Catalogs handle the highest level of separation, typically mapping to business domains, security requirements, or lifecycle stages. The schemas within those catalogs organize the interior, and volumes or tables store the actual data.

Organize data with catalogs

When you design your catalog structure, you're creating the foundation for **data governance** across your organization. Catalogs typically mirror organizational units or software development lifecycle scopes. Consider how your teams work and what data access boundaries matter most to your operations.

Environment-based isolation is one of the most common patterns. You might create separate catalogs for production, staging, and development environments. This separation prevents development queries from impacting production performance and ensures that experimental code never touches live customer data. With this pattern, your production catalog (`prod_catalog`) contains only validated, approved datasets, while your development catalog (`dev_catalog`) provides a safe space for testing and iteration.

Sensitivity-based isolation works well when compliance or privacy concerns drive your architecture. You might separate customer data from operational metrics, or isolate financial records from general business analytics. Each catalog can enforce different access policies, making it simple to grant broad access to public datasets while restricting sensitive information to authorized personnel.

Your catalog structure also affects **performance and costs**. Because each catalog can specify its own managed storage location, you can store frequently accessed production data in high-performance storage while archiving historical development data to more economical options. Regional compliance requirements become manageable too, since Unity Catalog metastores exist per region and catalogs inherit that regional isolation.

Create a catalog

Creating a catalog establishes the foundation for all data objects you'll add later. You must be a **metastore admin** or have the **CREATE CATALOG privilege** to create new catalogs. Once created, the catalog includes two automatically generated schemas: **default** for your general use and **information_schema** for system metadata.

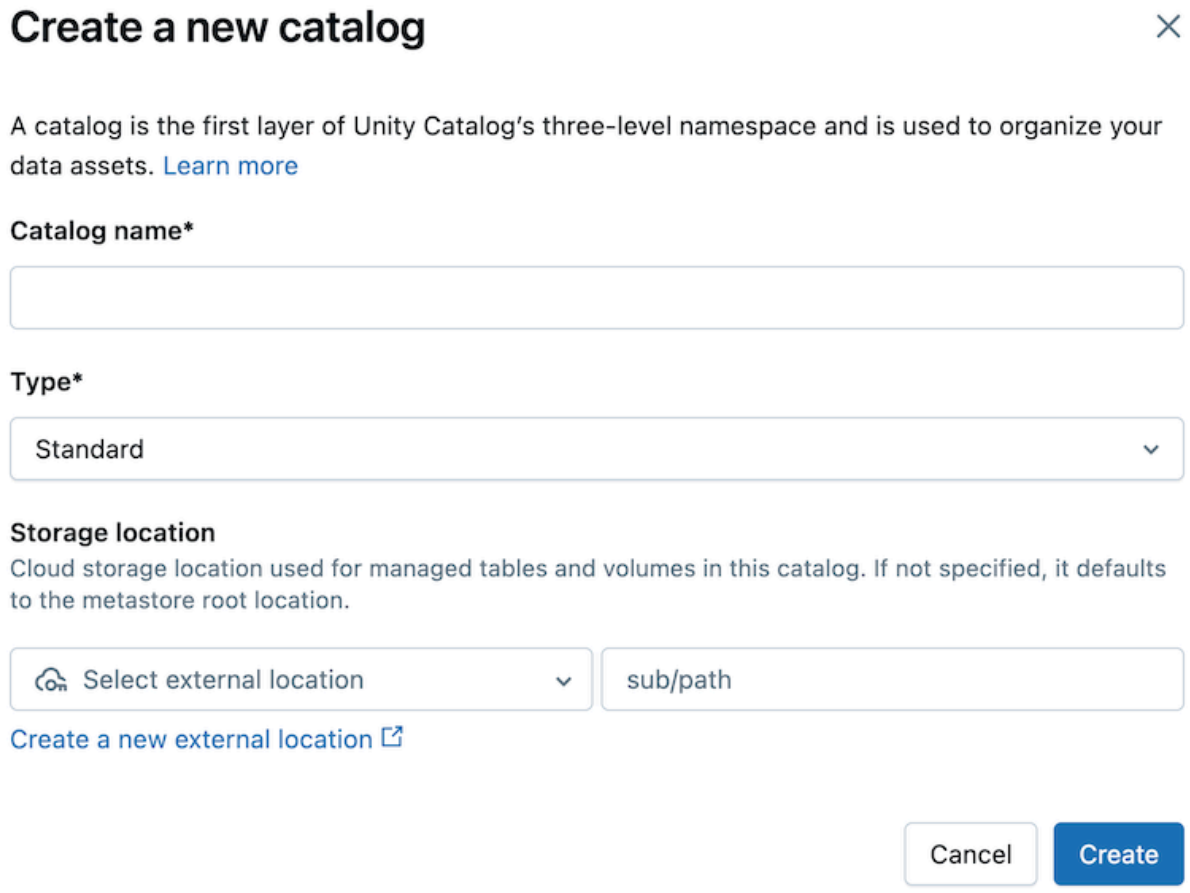
Using SQL, you can create a catalog with a simple command:

```
CREATE CATALOG IF NOT EXISTS dev_catalog
COMMENT 'Development environment for data engineering experiments';
```

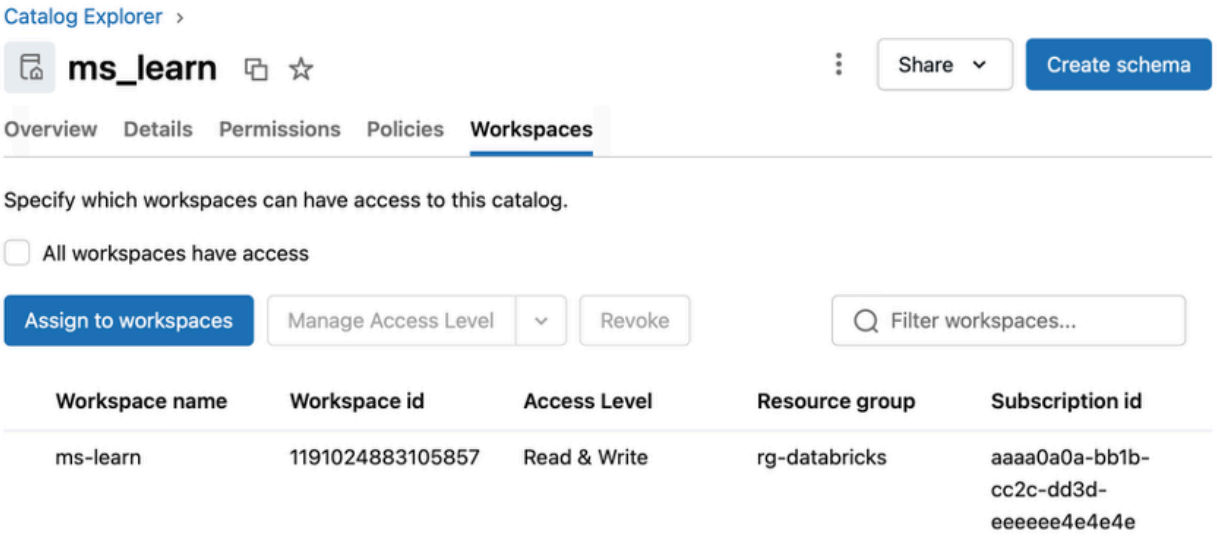
This creates a catalog named `dev_catalog` with a descriptive comment. The `IF NOT EXISTS` clause prevents errors if the catalog already exists, making your code more robust when run multiple times.

Using Catalog Explorer, you can create catalogs through the Azure Databricks UI:

1. Select **Catalog** from the left navigation.
2. Select **Create catalog**.
3. Enter a catalog name and select **Standard** as the type.
4. Optionally specify a managed storage location.
5. Select **Create**.



After creating the catalog, you configure **workspace bindings** to control which workspaces can access it. By default, new catalogs are accessible from all workspaces attached to your metastore. For production catalogs, you should restrict access to production workspaces only, ensuring that development environments can't accidentally query or modify production data.



Configure managed storage

While creating a catalog without specifying storage is possible, configuring a **managed storage location** is strongly recommended. Managed storage defines where Unity Catalog stores the data files for **managed tables** and **volumes** within your catalog. Without a catalog-level storage location, managed objects fall back to the metastore's default storage, which might not align with your organizational policies or regional requirements.

To specify managed storage, you need an **external location** already configured in Unity Catalog and the **CREATE MANAGED STORAGE privilege** on that location. The storage path must point to a cloud storage container, like an Azure Data Lake Storage Gen2 container, that meets your security and compliance requirements.

```
CREATE CATALOG IF NOT EXISTS prod_catalog
MANAGED LOCATION 'abfss://prod-data@mystorageaccount.dfs.core.windows.net/catalog-r
COMMENT 'Production catalog with dedicated ADLS Gen2 storage';
```

This command creates a catalog with dedicated production storage. All managed tables and volumes created in this catalog's schemas store their data files under this location, providing complete physical isolation from other catalogs. You can even specify a subpath within an external location, giving you flexibility to organize storage containers while maintaining centralized security credentials.

Managed storage serves two critical purposes. First, it provides **physical data isolation** by storing each catalog's data in separate cloud storage paths. When you drop a managed table, Unity Catalog can safely delete the underlying files after eight days without affecting data in other catalogs. Second, it helps you meet **compliance requirements** by ensuring sensitive data resides in specific storage accounts or regions that match your regulatory obligations.

If your metastore doesn't have a default storage location configured, you must specify managed storage when creating catalogs. This requirement ensures that Unity Catalog always knows where to place managed data files, preventing confusion and maintaining clear data governance boundaries.

Catalogs form the foundation of Unity Catalog's data organization model, providing the highest level of isolation and access control. By thoughtfully designing your catalog structure around environment boundaries or sensitivity levels, you create a governance framework that scales with your organization. Now that you understand how to create and configure catalogs, you're ready to organize the interior by adding schemas and data objects within those catalogs.

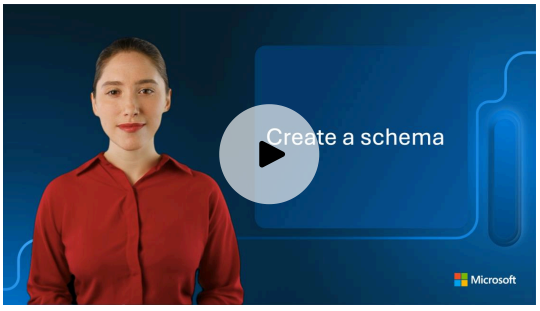
4. Create schema

https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/4-create-schema

Create schema

Completed

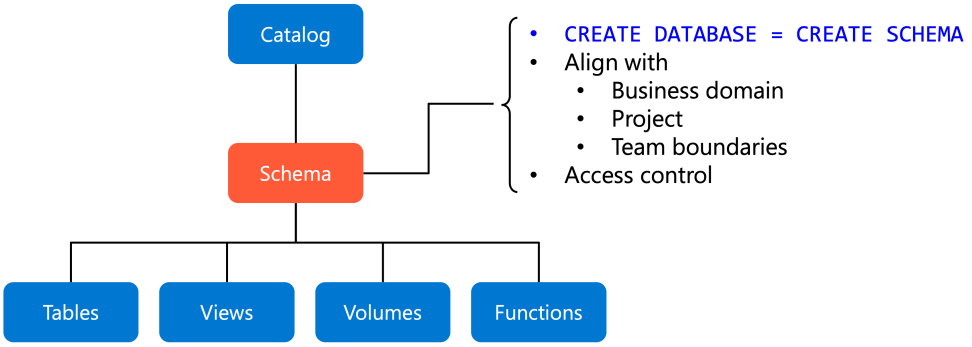
Your catalog provides the foundation for data isolation across environments, but within each catalog you need more detailed organization for your tables, views, and functions. Different teams work on different projects, departments manage distinct datasets, and various use cases require separate namespaces. **Schemas in Unity Catalog** give you this logical organization layer. They let you group related data assets within a catalog while maintaining clear boundaries and access controls.



What is a schema?

A **schema** is the **second level** in Unity Catalog's **three-layer namespace**: `catalog.schema.table`. While catalogs handle environment or sensitivity boundaries, schemas organize data within those catalogs into logical categories that typically represent a single use case, project, or team workspace.

Think of schemas as rooms within the house you built when creating your catalog. Each room serves a specific purpose and contains related items. Your production catalog might contain schemas for `customer_analytics`, `financial_reporting`, and `operations_metrics`, with each schema grouping the tables and views needed for that specific domain.



Schemas provide the organizational structure that makes your data **discoverable** and **manageable**. Instead of hundreds of tables scattered across a single namespace, you group related assets into meaningful categories. This structure helps data engineers find the datasets they need and understand how different data assets relate to each other.

In Azure Databricks, you might hear schemas called "databases." This is because `CREATE DATABASE` is an alias for `CREATE SCHEMA`. The terms are interchangeable in Databricks SQL, though "schema" aligns better with Unity Catalog's namespace hierarchy.

Organize data with schemas

When you design your schema structure within a catalog, consider how your teams work and what logical groupings make sense for your organization. Schemas typically align with **business domains**, **projects**, or **team boundaries**, creating clear ownership and access patterns.

```
Production Catalog: Department based schemas
├── marketing_analytics
├── financial_data
└── operations_metrics

Development Catalog: Project-based schemas
├── customer_churn_model
├── pricing_optimization
└── inventory_forecasting

Functional Organization: Stage-based Schemas
├── raw_data
├── cleansed_data
└── reporting_views
```

Department-based organization works well when different business units manage their own data. Within your production catalog, the marketing team might own a `marketing_analytics` schema, while the finance team maintains `financial_data`. Each department controls access to their own schema, granting read permissions to other teams as needed for cross-functional analytics.

Project-based organization suits environments where temporary initiatives need dedicated workspaces. Your development catalog might contain schemas like `customer_churn_model`,

`pricing_optimization`, and `inventory_forecasting`. When a project completes, you can archive or remove its schema without affecting other work. This pattern keeps experimental work organized and makes it easy to track which assets belong to which initiative.

Functional organization groups schemas by data processing stage or purpose. You might create schemas like `raw_data`, `cleansed_data`, and `reporting_views` within a catalog. This pattern mirrors data pipeline architecture, where each schema represents a stage in your data transformation workflow.

Your schema organization affects both **discoverability** and **access control**. Well-named schemas help users locate relevant datasets quickly. Clear schema boundaries make it straightforward to grant appropriate permissions—data engineers might have full access to the `staging` schema while analysts have read-only access to `curated_views`.

Create a schema

Creating a schema establishes a namespace for the tables, views, and functions you'll add later. You need **USE CATALOG** and **CREATE SCHEMA** permissions on the parent catalog. A metastore admin or the catalog owner can grant you these privileges.

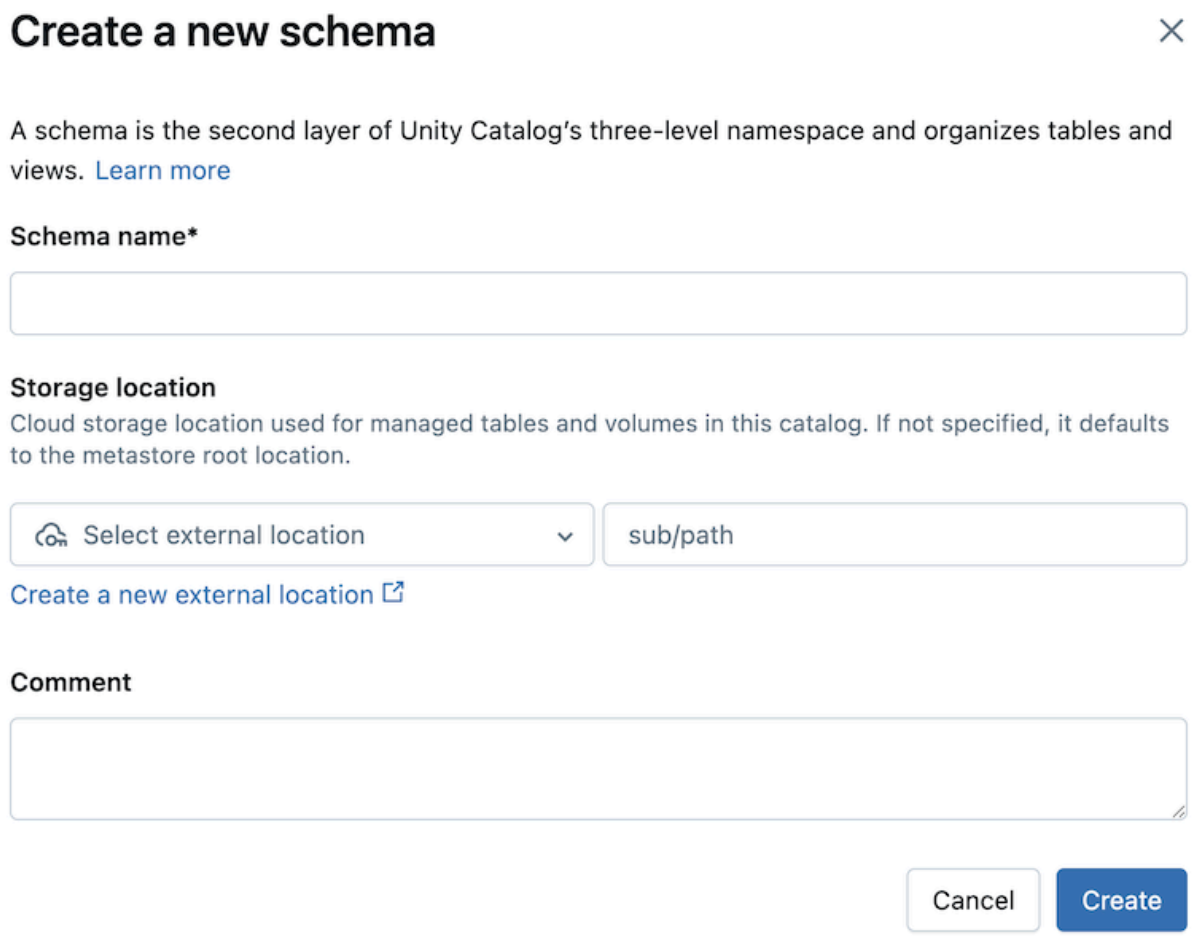
Using SQL, you create a schema with this command:

```
CREATE SCHEMA IF NOT EXISTS prod_catalog.customer_analytics
COMMENT 'Customer behavior and segmentation analysis';
```

This creates a schema named `customer_analytics` in the `prod_catalog` catalog. The `IF NOT EXISTS` clause prevents errors if the schema already exists, making your code safe to run repeatedly. The three-part namespace (`catalog.schema.table`) requires you to specify the catalog name, ensuring schemas end up in the correct container.

Using Catalog Explorer, you can create schemas through the Azure Databricks workspace UI:

1. Select **Catalog** from the left navigation.
2. Select the catalog where you want to create the schema.
3. Select **Create schema** from the catalog detail pane.
4. Enter a schema name and optional description.
5. Select **Create**.



After creating the schema, you grant privileges to control who can use it and create objects within it. At minimum, users need **USE SCHEMA** permission to see the schema and query its contents. To create tables or views in the schema, users need **CREATE TABLE** or other object-specific permissions.

Each new schema includes a system-generated **information_schema** that contains metadata about the schema's objects. You can query this metadata to discover tables, views, and their structure.

Configure managed storage

While not required, specifying a **managed storage location** for your schema gives you control over where Unity Catalog stores data files for managed tables created within the schema. Without a schema-level storage location, managed tables use the catalog's managed storage (or the metastore's default if the catalog has none specified).

To configure managed storage, you need an **external location** already set up in Unity Catalog and the **CREATE MANAGED STORAGE** privilege on that location. This lets you separate data for different schemas, even within the same catalog.

```
CREATE SCHEMA IF NOT EXISTS prod_catalog.financial_data
MANAGED LOCATION 'abfss://finance@mystorageaccount.dfs.core.windows.net/schema-data
```

```
COMMENT 'Financial transactions and reporting data with dedicated storage';
```

This command creates a schema with its own storage location in Azure Data Lake Storage Gen2. All managed tables created in this schema store their data files under this path, separate from other schemas in the catalog. This physical isolation helps you meet compliance requirements where financial data must reside in specific storage accounts with particular security configurations.

If you don't specify managed storage, your schema works the same way. Managed tables use the catalog's storage location, which simplifies management when physical isolation isn't required. You can always update the schema later to add a managed location if your requirements change.

5. Create tables and views

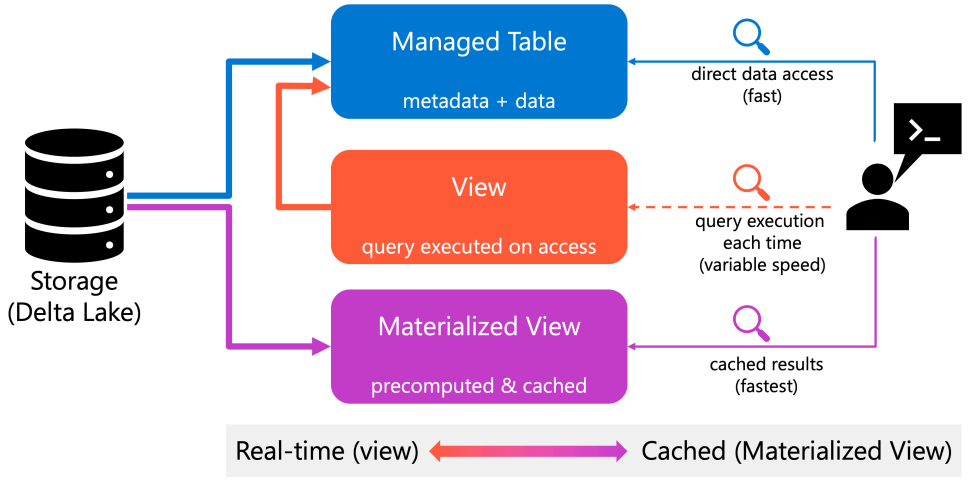
https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/5-create-tables-views-materialized

Create tables and views

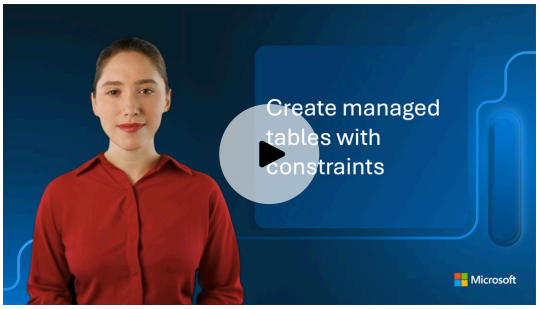
Completed

Tables and views form the foundation of your data organization in Unity Catalog. With tables, you persist data for long-term storage and analysis. With views, you simplify complex queries and control access to underlying data. With materialized views, you optimize query performance by precomputing and caching results. Understanding when and how to create each type helps you build efficient, maintainable data pipelines.

The following diagram illustrates how tables, views, and materialized views relate to each other and to the underlying data:



Create managed tables



Managed tables are the default table type in Unity Catalog. When you create a managed table, Azure Databricks manages both the table metadata and the underlying data files. This management includes automatic storage optimization, lifecycle management, and clean up when you drop the table.

To create a managed table, use the `CREATE TABLE` statement with column definitions. Unity Catalog uses a three-level namespace (`catalog.schema.table`), so you can either specify the full path or set your current catalog and schema context using `USE CATALOG` and `USE SCHEMA` statements. The following example creates a managed table for storing sales transactions:

```
-- Option 1: Use fully qualified name
CREATE TABLE production.sales.sales_transactions (
  transaction_id BIGINT,
  customer_id INT,
  product_name STRING,
  amount DECIMAL(10,2),
  transaction_date DATE
);

-- Option 2: Set current context first
USE CATALOG production;
USE SCHEMA sales;

CREATE TABLE sales_transactions (
  transaction_id BIGINT,
  customer_id INT,
  product_name STRING,
  amount DECIMAL(10,2),
  transaction_date DATE
);
```

Managed tables use the **Delta Lake** format by default, which provides ACID transactions, time travel, and schema enforcement. These features ensure data consistency and enable you to query historical versions of your data. When you drop a managed table, both the metadata and data files are deleted, preventing orphaned data in storage.

For production workloads, consider enabling features like **constraints** and **generated columns**. Constraints enforce data quality rules, while generated columns automatically compute values based on other columns. The following example shows a table with both features:

```
CREATE TABLE customer_orders (
  order_id BIGINT NOT NULL,
  customer_email STRING NOT NULL,
  order_total DECIMAL(10,2),
  tax_amount DECIMAL(10,2) GENERATED ALWAYS AS (order_total * 0.08),
  CONSTRAINT valid_total CHECK (order_total > 0)
);
```

Define primary keys and foreign keys

Primary keys and foreign keys help document table relationships and enable query optimizations in Unity Catalog. Available in Databricks Runtime 11.3 LTS and above (fully GA in Runtime 15.2+), these constraints are **informational only** and are **not enforced**. While they don't prevent invalid data, they provide valuable metadata for query optimization and data modeling.

To define a primary key, specify the `PRIMARY KEY` constraint during table creation. Primary key columns are implicitly `NOT NULL`. The following example creates a table with a single-column primary key:

```
CREATE TABLE customers (
  customer_id BIGINT NOT NULL,
  customer_name STRING,
  email STRING,
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

For composite primary keys spanning multiple columns, list all key columns in the constraint definition:

```
CREATE TABLE order_items (
  order_id BIGINT NOT NULL,
  item_id BIGINT NOT NULL,
  product_name STRING,
  quantity INT,
  price DECIMAL(10,2),
  CONSTRAINT order_items_pk PRIMARY KEY (order_id, item_id)
);
```

Foreign keys define relationships between tables by referencing primary keys in other tables. The following example creates a table with a foreign key constraint:

```
CREATE TABLE orders (
  order_id BIGINT NOT NULL PRIMARY KEY,
  customer_id BIGINT,
  order_date DATE,
  total_amount DECIMAL(10,2),
  CONSTRAINT orders_customers_fk FOREIGN KEY (customer_id) REFERENCES customers(cus
);
```

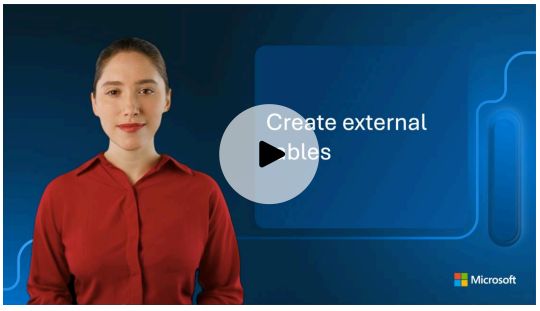
You can also add constraints to existing tables using `ALTER TABLE` statements:

```
ALTER TABLE orders
ADD CONSTRAINT orders_pk PRIMARY KEY (order_id);

ALTER TABLE order_items
ADD CONSTRAINT order_items_orders_fk FOREIGN KEY (order_id) REFERENCES orders(orde
```

While these constraints don't enforce referential integrity at write time, they enable the query optimizer to make better decisions about join strategies and query execution plans. This can improve performance for complex queries involving multiple table joins.

Create external tables



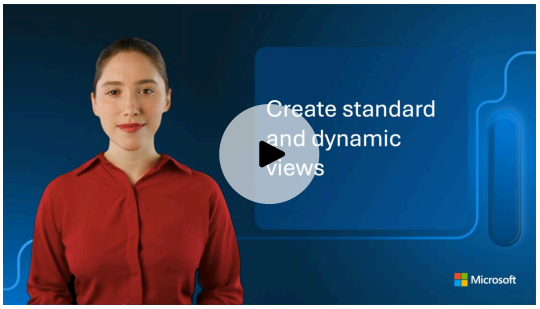
External tables reference data stored in external storage systems while registering metadata in Unity Catalog. Unlike managed tables, dropping an external table only removes the metadata—the underlying data files remain in their original location. This separation makes external tables useful when you need to access data managed by other systems or when you want to share data across multiple workspaces.

To create an external table, specify a `LOCATION` clause pointing to your data. The location must be protected by an external location configured in Unity Catalog. The following example creates an external table over CSV files:

```
CREATE EXTERNAL TABLE archived_sales
USING CSV
LOCATION 'abfss://container@storage.dfs.core.windows.net/sales/archive'
OPTIONS (
  header 'true',
  inferSchema 'true'
);
```

External tables work best when you need to query data in its original format without moving it, or when multiple systems need access to the same data. However, you lose some benefits of managed tables, such as automatic cleanup and certain Delta Lake optimizations. For new data pipelines, prefer managed tables unless you have a specific requirement for external storage.

Create standard views



Views provide a virtual layer over your tables, defined by a query that executes each time you access the view. Standard views don't store data—they compute results on demand by running the underlying query. This design makes views ideal for simplifying complex joins, encapsulating business logic, or providing consistent interfaces to frequently accessed data.

To create a view, use the `CREATE VIEW` statement with a `SELECT` query. The following example creates a view that combines customer and order information:

```
CREATE VIEW customer_order_summary AS
SELECT
  c.customer_id,
  c.customer_name,
  c.region,
  COUNT(o.order_id) AS total_orders,
  SUM(o.order_total) AS total_spent
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.customer_name, c.region;
```

Views update automatically when underlying tables change, ensuring you always query current data. This behavior differs from materialized views, which cache results and require explicit refreshes. However, complex views can be slow if they aggregate large datasets or perform expensive joins. For frequently accessed complex queries, consider materialized views instead.

You can layer views on top of other views to create multi-level abstractions. This approach helps organize complex logic into manageable pieces. At the same time, too many view layers can make troubleshooting difficult and impact performance. Balance abstraction with clarity and performance needs.

Create dynamic views for access control

Dynamic views extend standard views with fine-grained security controls. With dynamic views, you apply **row-level filters** and **column-level masks** based on the user querying the view. This capability lets you share data while controlling what different users can see, without duplicating tables or creating multiple views.

To create a dynamic view with column masking, use the `CASE` statement combined with the `is_account_group_member()` function. The following example shows how to redact email addresses for users not in the auditors group:

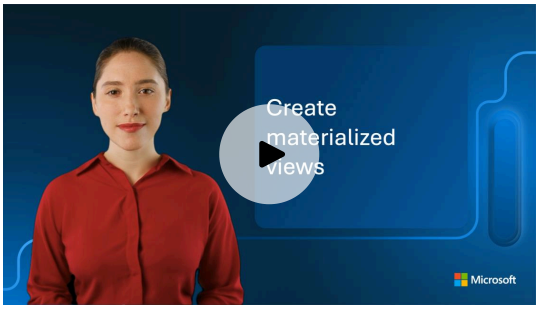
```
CREATE VIEW sales_redacted AS
SELECT
  user_id,
  CASE WHEN
    is_account_group_member('auditors') THEN email
    ELSE 'REDACTED'
  END AS email,
  country_region,
  product,
  total
FROM sales_raw;
```

For row-level security, use a `WHERE` clause with conditional logic. The following example restricts high-value transactions to managers only:

```
CREATE VIEW sales_filtered AS
SELECT
  user_id,
  country_region,
  product,
  total
FROM sales_raw
WHERE
  CASE
    WHEN is_account_group_member('managers') THEN TRUE
    ELSE total <= 1000000
  END;
```

Dynamic views require SQL warehouses, standard access mode compute, or dedicated access mode compute on Databricks Runtime 15.4 LTS or above. These security controls ensure sensitive data remains protected while still enabling broad access to analytics. Without dynamic views, you would need to create separate tables or views for each access level, increasing maintenance overhead.

Create materialized views



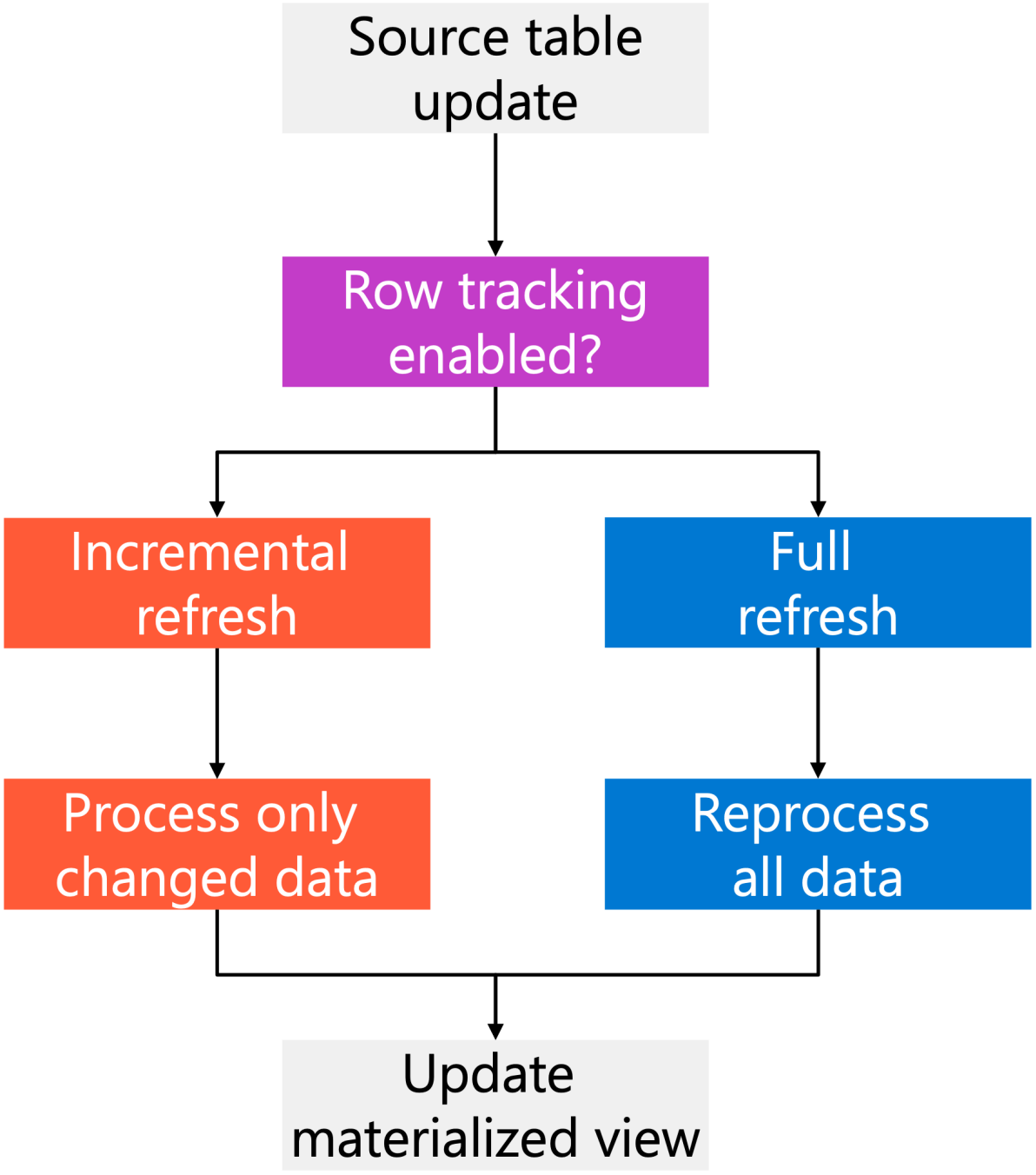
Materialized views precompute and cache query results, storing them as a Delta table. Unlike standard views that recalculate results on every query, materialized views return cached data, making them significantly faster for complex aggregations and frequently accessed queries. You refresh materialized views manually or on a schedule to reflect changes in underlying data.

To create a materialized view, use the `CREATE MATERIALIZED VIEW` statement. The following example creates a materialized view for daily sales summaries:

```
CREATE MATERIALIZED VIEW daily_sales_summary AS
SELECT
  transaction_date,
  COUNT(*) AS transaction_count,
  SUM(amount) AS total_sales,
  AVG(amount) AS average_sale
FROM sales_transactions
GROUP BY transaction_date;
```

Materialized views support two refresh strategies: **incremental** and **full**. Incremental refresh processes only changed data, which is far more efficient than reprocessing everything. To support incremental refresh, your source tables must be Delta tables with **row tracking** enabled:

The following diagram shows how materialized views handle different refresh strategies:



To enable row tracking on your source tables:

```
ALTER TABLE sales_transactions
SET TBLPROPERTIES (delta.enableRowTracking = true);
```

Note

Row tracking requires Databricks Runtime 14.1 or above. Enabling row tracking on existing tables automatically assigns row IDs to all rows, which can take significant time for large tables

> and creates multiple new table versions. This operation upgrades the table protocol version and can't be reversed.

You can schedule automatic refreshes using the `SCHEDULE` clause when creating the materialized view, or trigger refreshes when upstream data changes. The following example schedule daily refreshes:

```
CREATE MATERIALIZED VIEW daily_sales_summary
SCHEDULE EVERY 1 DAY
AS
SELECT
  transaction_date,
  COUNT(*) AS transaction_count,
  SUM(amount) AS total_sales
FROM sales_transactions
GROUP BY transaction_date;
```

Materialized views excel at improving dashboard performance and reducing compute costs for frequently run analytics. Consider materialized views when queries take more than a few seconds to run, or when multiple users query the same aggregated data repeatedly. For data that changes constantly and requires real-time accuracy, standard views might be more appropriate despite slower performance.

Best practices for tables and views

When creating tables, start with **managed tables** unless you have a specific need for external storage. Managed tables provide better integration with Unity Catalog features, automatic optimization, and simpler lifecycle management. Enable row tracking on Delta tables that serve as sources for materialized views to support efficient incremental refreshes.

For **views**, keep the underlying queries focused and avoid creating too many layers. While layering views provides logical organization, it can make debugging difficult and hide performance issues. Document the purpose of each view and its intended audience to help other team members understand your data model.

Use **dynamic views** to implement data security policies at the view level rather than duplicating tables for different user groups. This approach centralizes security logic and reduces data redundancy. Combine dynamic views with Unity Catalog permissions to create a comprehensive data governance strategy.

When choosing between standard and materialized views, consider query performance and data freshness requirements. Use standard views for queries that run quickly or when you need real-time data. Use **materialized views** for expensive aggregations or when eventual consistency is acceptable. Monitor the refresh behavior of materialized views to ensure they update incrementally rather than fully reprocessing data.

Apply **liquid clustering** to tables and materialized views instead of traditional partitioning for better query performance and easier maintenance. Liquid clustering automatically optimizes data layout based on query patterns, while partitioning requires manual configuration and can create skewed partitions. Use the `CLUSTER BY` clause when creating tables or materialized views to enable this optimization.

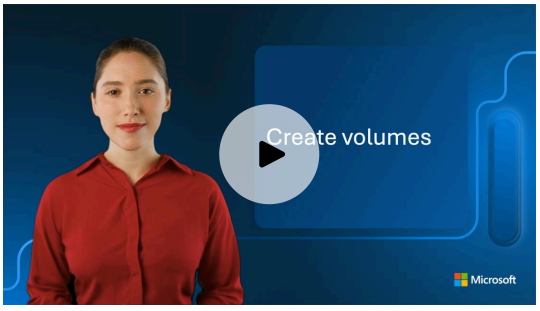
6. Create volumes

https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/6-create-volumes

Create volumes

Completed

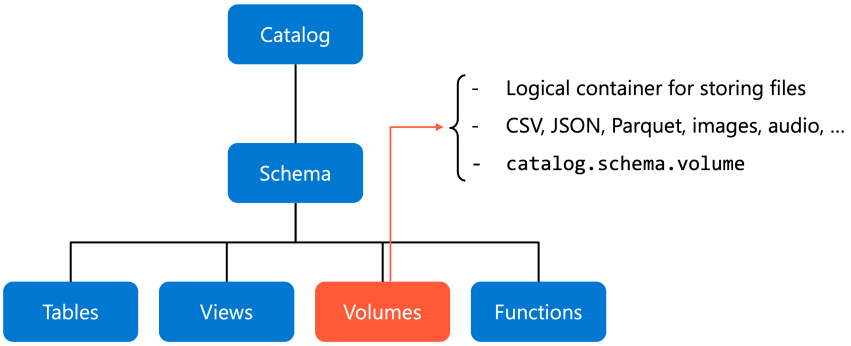
Your data engineering team needs to store machine learning models, CSV files for ingestion, and image files for analytics workloads. You already organized your data using catalogs and schemas, but you need a governed way to manage these nontabular files in cloud object storage. **Unity Catalog volumes** provide the solution by extending governance to files while maintaining the same security model you use for tables.



What are volumes?

A **volume** is a Unity Catalog object that represents a logical container for files stored in cloud object storage. Think of it as the final piece in organizing your data house—after building the foundation with catalogs and organizing the interior with schemas, volumes provide the space to store your raw files.

Volumes sit at the third level of Unity Catalog's namespace structure: `catalog.schema.volume`. Unlike tables that govern structured data, volumes govern files of any format—CSV, JSON, Parquet, images, audio files, or machine learning artifacts.



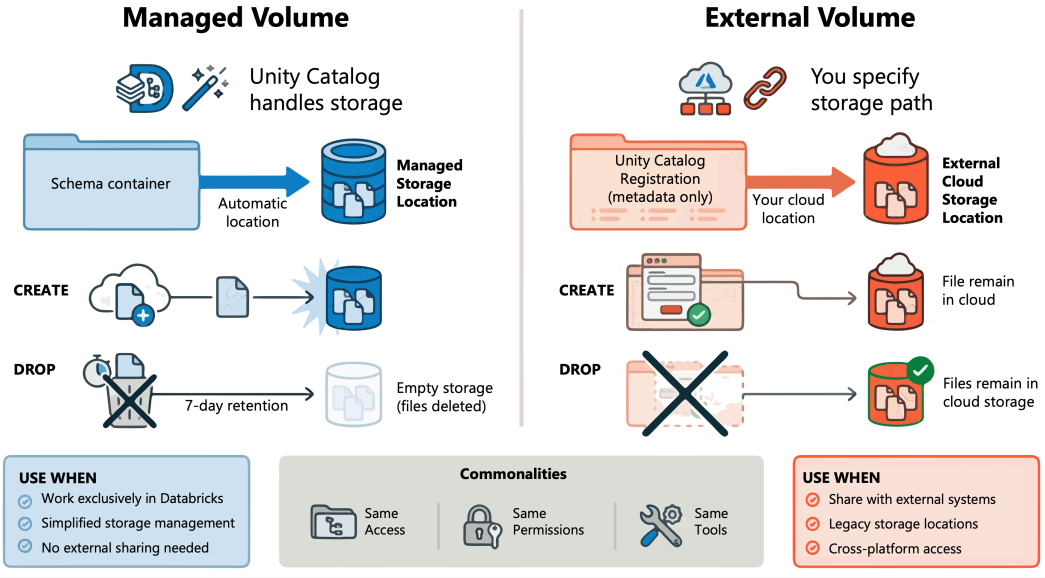
With Unity Catalog's three-layer hierarchy, understanding where volumes fit becomes essential. Volumes handle file-based data governance, providing path-based access to files while maintaining centralized security controls. This separation allows you to manage structured data through tables and unstructured data through volumes, both under the same governance framework.

Choose between managed and external volumes

When you create a volume, you choose between **managed** and **external** volumes. This decision affects where your files are stored and what happens when you delete the volume.

Managed volumes offer the simplest approach. Unity Catalog handles storage location and lifecycle management automatically. When you create a managed volume, Azure Databricks stores files in the managed storage location associated with your schema. When you drop a managed volume, Unity Catalog marks the files for deletion after a seven-day retention period.

External volumes provide governance for existing cloud storage locations. You specify the storage path when creating the volume, and the files remain in that location throughout the volume's lifecycle. When you drop an external volume, the files remain in cloud storage—only the Unity Catalog registration is removed.



Consider managed volumes when you work exclusively within Azure Databricks and want simplified storage management. Your data engineering pipelines can write files directly to volumes without worrying about underlying storage paths or credentials.

Consider external volumes when you need to share data between Azure Databricks and other systems. Perhaps your machine learning models must be accessed by external applications, or you need to add governance to files already produced by legacy systems. External volumes allow you to maintain existing storage locations while adding Unity Catalog's security controls.

The technical implementation remains nearly identical regardless of your choice. You access files using the same path format, apply the same permissions, and use the same tools. The main difference lies in storage control and data lifecycle management.

Create a volume

Creating a volume establishes a governed storage location for your files. You must have **CREATE VOLUME** privilege on the schema and **USE** privileges on both the schema and catalog. For external volumes, you also need **CREATE EXTERNAL VOLUME** privilege on the target external location.

Using SQL, create a managed volume with this syntax:

```
CREATE VOLUME IF NOT EXISTS dev_catalog.bronze_schema.landing_files
COMMENT 'Landing area for CSV files from external systems';
```

This creates a volume named `landing_files` in the specified schema. The `IF NOT EXISTS` clause prevents errors if the volume already exists, making your code resilient to repeated execution.

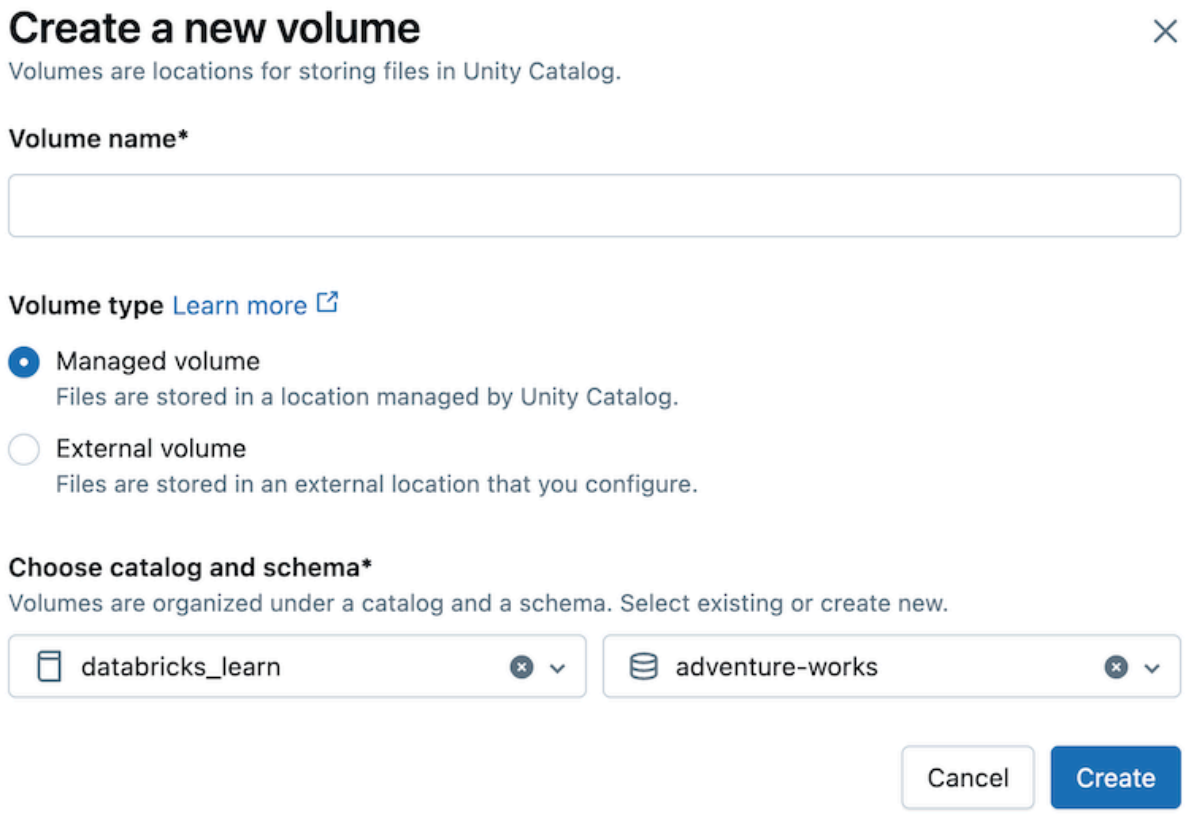
For external volumes, add the `LOCATION` clause to specify the cloud storage path:

```
CREATE EXTERNAL VOLUME prod_catalog.silver_schema.ml_models
LOCATION 'abfss://models@mystorageaccount.dfs.core.windows.net/production/';
```

The location must point to a path within an external location that Unity Catalog already governs. This ensures that Unity Catalog maintains security control even when the volume references existing storage.

Using Catalog Explorer, create volumes through the UI:

1. Select **Catalog** from the left navigation.
2. Navigate to the target schema and select it.
3. Select **Create** > **Volume**.
4. Enter a volume name.
5. Choose **Managed** or **External** as the volume type.
6. For external volumes, select an external location and specify the subdirectory path.
7. Select **Create**.



After creation, your volume is ready to store files. The volume inherits permissions from its parent schema, though you can grant specific permissions to individual users or groups as needed.

Access files in volumes

Once you create a volume, you access files using a standard path format that works across all Azure Databricks tools and languages. The path follows this pattern:

```
/Volumes/<catalog>/<schema>/<volume>/<path>/<filename>
```

This POSIX-style path works with Apache Spark, pandas, SQL, and other frameworks without requiring cloud storage credentials or connection strings. For example, to read a CSV file stored in your volume:

```
df = spark.read.format("csv").load("/Volumes/dev_catalog/bronze_schema/landing_file
```

The same path works in pandas:

```
import pandas as pd
df = pd.read_csv('/Volumes/dev_catalog/bronze_schema/landing_files/data.csv')
```

You can also query files directly using SQL:

```
SELECT * FROM csv.`/Volumes/dev_catalog/bronze_schema/landing_files/data.csv`;
```

For file management operations, use `dbutils.fs` commands in notebooks:

```
# List files in a volume
dbutils.fs.ls("/Volumes/dev_catalog/bronze_schema/landing_files")

# Copy a file
dbutils.fs.cp(
    "/Volumes/dev_catalog/bronze_schema/landing_files/source.csv",
    "/Volumes/dev_catalog/silver_schema/processed/destination.csv"
)
```

External volumes support an additional access pattern. Users with appropriate permissions can access files using cloud storage URIs directly, such as `abfss://container@account.dfs.core.windows.net/path/file.csv`. However, Unity Catalog still governs this access through volume permissions, not external location permissions.

Manage volume permissions

Volume permissions control who can read and write files. Unity Catalog provides four key privileges for volumes:

- **READ VOLUME**: Allows listing and reading files in the volume
- **WRITE VOLUME**: Allows creating, updating, and deleting files
- **CREATE VOLUME**: Allows creating new volumes in a schema
- **CREATE EXTERNAL VOLUME**: Allows creating external volumes using an external location

To grant read access to a data engineers group:

```
GRANT READ VOLUME ON VOLUME dev_catalog.bronze_schema.landing_files TO `data-engine
```

To grant full read and write access to data engineers:

```
GRANT READ VOLUME, WRITE VOLUME ON VOLUME dev_catalog.bronze_schema.landing_files
```

Permissions cascade from parent objects. If you grant READ VOLUME at the catalog level, users can read files from all volumes in that catalog. This makes it simple to set up broad access policies while restricting sensitive data through more specific grants.

> **Note**
>
> Unity Catalog volumes don't support folder-level or subdirectory-level ACLs. Permissions apply to the entire volume. If you need different access controls for different sets of files, create separate volumes for each security boundary.

Remember that reading or writing files requires both volume permissions and the necessary `USE` privileges on the parent catalog and schema. Unity Catalog enforces this hierarchy to maintain consistent access control across your data assets.

7. Implement DDL operations

https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/7-implement-data-definition-language-operations

Implement DDL operations

Completed

Your data engineering team needs to manage complex data transformations, enforce data quality rules, and maintain streaming pipelines across multiple environments. You need **reusable functions** to standardize calculations, **stored procedures** to orchestrate multi-step operations, and **streaming tables** to process real-time data. Unity Catalog's **Data Definition Language (DDL) operations** provide the tools to create and modify these objects while maintaining governance and version control.

Create functions for reusable logic

Functions encapsulate business logic that you can reuse across queries and transformations. Unity Catalog supports both **scalar functions** that return single values and **table-valued functions** that return entire result sets. You can write functions in **SQL** or **Python**, depending on your team's expertise and the complexity of the logic.

When you create a function in Unity Catalog, it becomes a governed object that you can share across teams and workspaces. With Unity Catalog, you define functions once and invoke them anywhere within your data platform. The catalog tracks function ownership, dependencies, and usage patterns, providing visibility into how your organization uses shared logic.

SQL scalar functions work well for simple calculations and transformations. You define the function with parameters, specify the return type, and provide the SQL expression that computes the result:

```
CREATE FUNCTION dev_catalog.finance_schema.calculate_discount(price DOUBLE, discou
RETURNS DOUBLE
COMMENT 'Calculate discounted price'
RETURN price * (1 - discount_rate);
```

This function accepts two numeric parameters and returns the discounted price. You can now call this function from any query within your Unity Catalog environment, ensuring consistent discount calculations across all reports and analytics.

Python functions handle more complex logic that requires external libraries or advanced data processing. Unity Catalog executes Python functions using serverless or pro SQL warehouses:

```
CREATE FUNCTION dev_catalog.analytics_schema.normalize_text(input_text STRING)
RETURNS STRING
LANGUAGE PYTHON
COMMENT 'Normalize text using Python'
AS $$
  return input_text.lower().strip() if input_text else None
$$;
```

Table-valued functions return complete result sets rather than scalar values. This approach works well when you need to transform arrays into rows or generate multiple records from a single input:

```
CREATE FUNCTION dev_catalog.utils_schema.split_to_rows(input_array ARRAY<STRING>)
RETURNS TABLE(value STRING)
RETURN SELECT explode(input_array) AS value;
```

You can then query this function like a table: `SELECT * FROM dev_catalog.utils_schema.split_to_rows(array('a', 'b', 'c'))`.

You can also create table-valued functions using **Python**, which is useful when you need more complex processing logic:

```
CREATE FUNCTION dev_catalog.utils_schema.generate_sequence(start INT, end INT)
RETURNS TABLE(num INT)
LANGUAGE PYTHON
COMMENT 'Generate a sequence of numbers from start to end'
AS $$
  return [(i,) for i in range(start, end + 1)]
$$;
```

Temporary functions exist only within your current session and don't persist in the catalog. This approach works well for one-time analyses or testing new logic before creating a permanent function:

```
CREATE TEMPORARY FUNCTION calculate_tax(amount DOUBLE)
RETURNS DOUBLE
RETURN amount * 0.08;
```

Temporary functions don't require catalog or schema qualifiers because they exist only in your session scope.

Create procedures for multi-step operations

SQL stored procedures let you define sequences of SQL statements that run together, with support for variables, control flow, and error handling. Procedures become particularly valuable when you need to orchestrate complex data operations that involve multiple steps, conditional logic, or iterative processing.

With procedures, you can encapsulate entire workflows—like data validation, transformation, and loading—into a single callable unit. You create procedures using the `CREATE PROCEDURE` statement, which requires Unity Catalog and Databricks Runtime 17.0 or above:

```
CREATE PROCEDURE dev_catalog.etl_schema.refresh_customer_summary(
  IN source_date DATE,
  OUT rows_processed INT
)
LANGUAGE SQL
SQL SECURITY INVOKER
COMMENT 'Refresh customer summary table for specified date'
BEGIN
  DECLARE row_count INT;

  DELETE FROM dev_catalog.analytics_schema.customer_summary
  WHERE summary_date = source_date;

  INSERT INTO dev_catalog.analytics_schema.customer_summary
  SELECT customer_id, summary_date, SUM(amount) as total
  FROM dev_catalog.sales_schema.transactions
  WHERE transaction_date = source_date
  GROUP BY customer_id, summary_date;

  SET row_count = (SELECT COUNT(*) FROM dev_catalog.analytics_schema.customer_summa
                   WHERE summary_date = source_date);
  SET rows_processed = row_count;
END;
```

This procedure accepts an input date parameter and returns the number of rows processed through an output parameter. You call procedures using the `CALL` statement:

```
CALL dev_catalog.etl_schema.refresh_customer_summary(DATE '2024-01-15', :rows);
```

Procedures support **IN** parameters for inputs, **OUT** parameters for outputs, and **INOUT** parameters that serve both purposes. The `SQL SECURITY INVOKER` clause specifies that the procedure runs with

the permissions of the user calling it, ensuring proper access control.

Modify catalog ownership and properties

As your data organization evolves, you need to transfer ownership of catalogs or modify their properties. The `ALTER CATALOG` statement provides these capabilities, letting you change catalog owners, apply tags for metadata management, or enable predictive optimization.

Transferring ownership ensures that the right team or principal manages each catalog. You might transfer ownership when reorganizing teams or delegating catalog management to a different group:

```
ALTER CATALOG dev_catalog SET OWNER TO `data_engineering_team`;
```

This command transfers ownership to a group named `data_engineering_team`. Only the current owner, a metastore admin, or a user with the `MANAGE` privilege can transfer ownership.

Applying tags helps you organize and classify catalogs according to your organization's taxonomy. Tags provide searchable metadata that supports governance and discovery:

```
ALTER CATALOG dev_catalog SET TAGS (
  environment = 'development',
  cost_center = 'engineering',
  data_classification = 'internal'
);
```

Enabling predictive optimization automates maintenance tasks like compaction and vacuum operations. Unity Catalog monitors usage patterns and applies optimizations when appropriate:

```
ALTER CATALOG prod_catalog ENABLE PREDICTIVE OPTIMIZATION;
```

Tables within the catalog inherit this setting unless you explicitly override it at the table level.

Modify table structure and constraints

As your data requirements change, you need to modify table schemas without rebuilding entire datasets. The `ALTER TABLE` statement supports adding columns, renaming columns, dropping columns, managing constraints, and changing column types—all while preserving your existing data.

Adding columns extends your table schema to accommodate new data fields. When you add a column to an existing Delta table, Unity Catalog populates it with `NULL` values for existing rows:

```
ALTER TABLE dev_catalog.sales_schema.orders
ADD COLUMN shipping_carrier STRING
COMMENT 'Name of the shipping carrier';
```

Renaming columns updates column names without moving data. This operation requires Delta Lake column mapping, which you enable by setting table properties:

```
ALTER TABLE dev_catalog.sales_schema.orders
RENAME COLUMN customer_email TO email_address;
```

Dropping columns removes columns from your table schema. You must drop any dependent constraints or generated columns first:

```
ALTER TABLE dev_catalog.sales_schema.orders
DROP COLUMN temporary_field;
```

Changing column types adapts your schema to new requirements. You can widen types (like `INT` to `BIGINT`) or change compatible types while Unity Catalog handles data conversion:

```
ALTER TABLE dev_catalog.sales_schema.orders
ALTER COLUMN quantity TYPE BIGINT;
```

These operations work on Unity Catalog tables and require the `MODIFY` privilege.

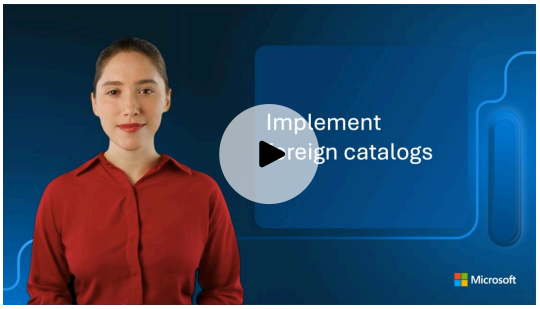
8. Implement foreign catalog

https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/8-implement-foreign-catalog

Implement foreign catalog

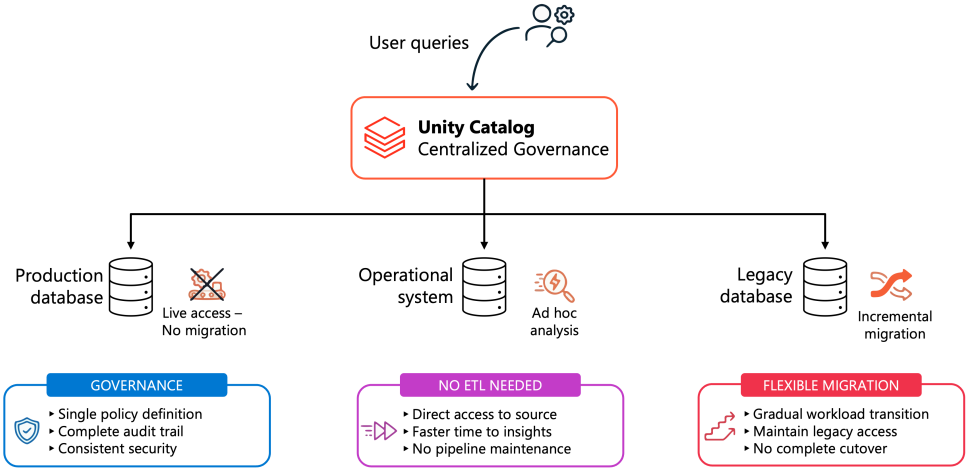
Completed

Your analytics team needs immediate access to customer transaction data stored in a PostgreSQL database. Moving terabytes of operational data into Azure Databricks for exploratory analysis would take weeks and consume significant resources. At the same time, your organization requires **centralized governance** for all data access, regardless of where the data resides. **Foreign catalogs in Unity Catalog** solve this challenge by enabling you to query external databases directly while maintaining Unity Catalog's access control, lineage tracking, and audit capabilities.



Why implement foreign catalogs?

Foreign catalogs extend Unity Catalog's governance to external data systems without requiring data migration. This approach becomes essential when you need to balance operational efficiency with security and compliance requirements.



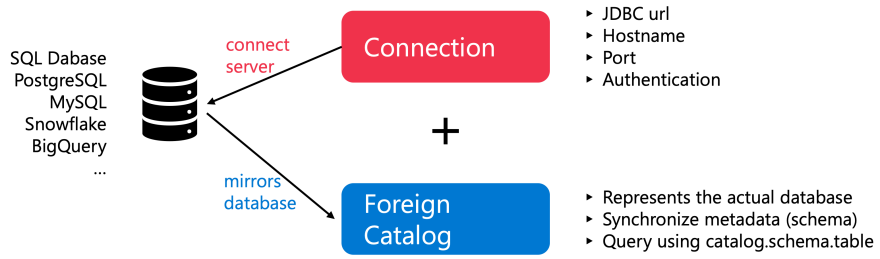
Governance and auditability remain consistent across your data estate. With foreign catalogs, your security team defines access policies once in Unity Catalog, and those policies apply whether users query data in Databricks-managed tables or external databases. Every query against foreign data appears in Unity Catalog's audit logs, providing complete visibility into who accessed what data and when. This centralized governance eliminates the complexity of managing permissions across multiple systems.

External data access without ETL accelerates time to insights. You can run proof-of-concept analyses on production databases, create ad hoc reports from live operational systems, or validate data quality before committing to a full migration. The data stays in its original location, avoiding the time and cost of building and maintaining ETL pipelines for exploratory work.

Incremental migration support provides flexibility during transitions to Unity Catalog. Organizations moving from legacy systems can implement foreign catalogs to maintain access to data that hasn't migrated yet, allowing workloads to transition gradually rather than requiring a complete cutover.

Understand connections and foreign catalogs

A **connection** and a **foreign catalog** work together to enable external data access, but they serve different purposes in the architecture. Understanding this relationship is essential before you begin implementation.



A **connection** stores the technical details needed to reach an external database system. Think of it as a registered credential that specifies where the database lives and how to authenticate to it. The connection contains the JDBC URL, hostname, port, and authentication credentials for systems like PostgreSQL, MySQL, Snowflake, or BigQuery. You create the connection once and can use it to establish multiple foreign catalogs if needed.

A **foreign catalog** represents the actual database you want to query. It mirrors the structure of an external database within Unity Catalog's namespace, making external schemas and tables accessible through standard SQL queries. When you create a foreign catalog, you specify which connection to use and which external database to mirror. Unity Catalog then synchronizes the metadata, allowing you to reference external tables using the familiar `catalog.schema.table` naming pattern.

This separation between connection and foreign catalog provides important flexibility. A single connection to your PostgreSQL server might support multiple foreign catalogs, each representing a different database on that server. The connection handles authentication, while the foreign catalog determines what data becomes visible in Unity Catalog.

Configure a connection to the external database

Before you can create a foreign catalog, you must establish a connection that defines how Azure Databricks reaches your external database. The connection stores credentials securely and validates network connectivity to the target system.

Prerequisites must be in place before creating a connection. You need either the **CREATE CONNECTION** privilege on the Unity Catalog metastore or metastore admin permissions. Your Azure Databricks compute must use Runtime 13.3 LTS or above with Standard or Dedicated access mode. Network connectivity between your Azure Databricks workspace and the external database is essential; firewalls and network security groups must allow traffic on the database port.

Using SQL, you create a connection by specifying the database type and configuration options. This example creates a connection to a PostgreSQL database:

```
CREATE CONNECTION postgresql_prod TYPE postgresql
OPTIONS (
  host 'prod-db.example.com',
  port '5432',
  user secret ('prod-secrets','postgres-user'),
  password secret ('prod-secrets','postgres-password')
);
```

This command establishes a connection named `postgresql_prod` that points to a PostgreSQL server. The `secret` function references credentials stored in Databricks secrets rather than embedding passwords directly in the SQL statement, following security best practices. Each database type (MySQL, Snowflake, BigQuery) requires specific options appropriate to that system.

For **Azure SQL Database**, you would create a connection like this:

```
CREATE CONNECTION azuresql_prod TYPE SQLSERVER
OPTIONS (
  host 'myserver.database.windows.net',
  port '1433',
  user secret ('prod-secrets','azuresql-user'),
  password secret ('prod-secrets','azuresql-password')
);
```

Using Catalog Explorer, you can create connections through the Azure Databricks interface:

1. In the navigation menu, select **Catalog**.
2. Select the **Add** icon and choose **Add a connection**.
3. Enter a name for your connection, such as `postgresql_prod`.
4. Select the connection type (for example, PostgreSQL, MySQL, or Snowflake).
5. Provide the required connection details including host, port, and credentials.
6. (Optional) Select **Test connection** to verify connectivity.
7. Select **Create connection**.

Validation confirms that Azure Databricks can reach your database. After creating the connection, test it by attempting a simple metadata query. If the connection fails, verify network rules, credentials, and that the database server accepts connections from Azure Databricks IP addresses.

Create a foreign catalog using the connection

With a validated connection in place, you're ready to create the foreign catalog that makes external tables accessible through Unity Catalog. This step establishes the mapping between your external database and Unity Catalog's namespace.

Permissions required include the **CREATE CATALOG** privilege on the metastore and either ownership of the connection or the **CREATE FOREIGN CATALOG** privilege on that specific connection. These granular permissions allow you to control who can expose external data through Unity Catalog.

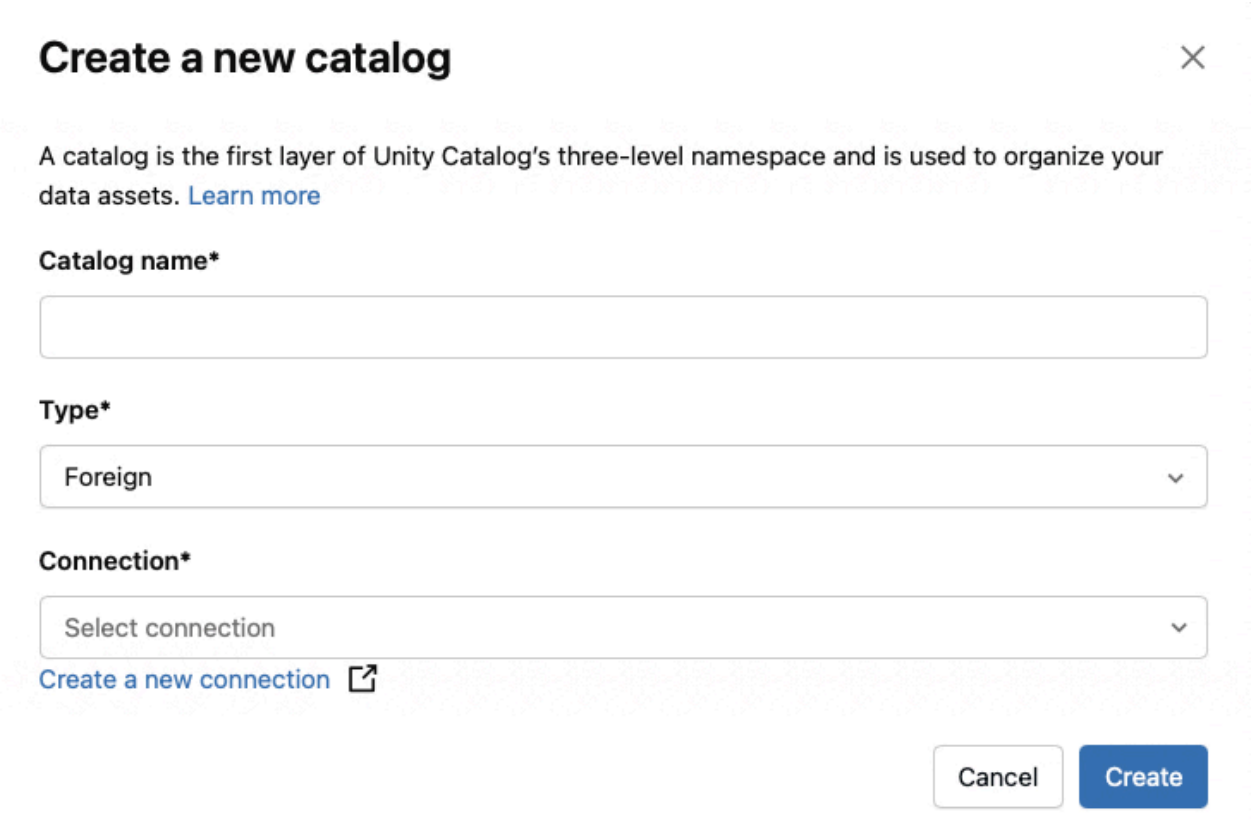
Using SQL, you create a foreign catalog by referencing the connection and specifying which external database to mirror:

```
CREATE FOREIGN CATALOG IF NOT EXISTS prod_customer_data
USING CONNECTION postgresql_prod
OPTIONS (database 'customers');
```

This command creates a catalog named `prod_customer_data` that mirrors the `customers` database from your PostgreSQL server. The `IF NOT EXISTS` clause prevents errors when running the script multiple times. Unity Catalog immediately synchronizes metadata from the external database, making its schemas and tables visible in your workspace.

Using Catalog Explorer, you can create the foreign catalog during connection setup or afterward:

1. Select **Catalog** and then select **Create a catalog**.
2. Choose **Foreign catalog** as the catalog type.
3. Enter a name like `prod_customer_data`.
4. Select the connection you created earlier.
5. Specify the external database name (for example, `customers`).
6. Select **Create**.



Metadata synchronization happens automatically each time you interact with the foreign catalog. When you query `SELECT * FROM prod_customer_data.public.transactions`, Unity Catalog refreshes its view of the external schema structure before executing the query. This ensures you

always see current metadata, even if database administrators add new tables or columns in the external system.

While automatic synchronization is optimal for most workloads, it can impact performance for high-frequency queries or when external systems are accessed by tools outside Databricks. For these scenarios, you can manually refresh metadata on a schedule using the `REFRESH FOREIGN` command. This approach allows queries to run faster using cached metadata rather than refreshing on every query. You can refresh at different levels of granularity:

```
-- Refresh an entire catalog
REFRESH FOREIGN CATALOG prod_customer_data;

-- Refresh a specific schema
REFRESH FOREIGN SCHEMA prod_customer_data.public;

-- Refresh a specific table
REFRESH FOREIGN TABLE prod_customer_data.public.transactions;
```

Schedule these refresh operations using Lakeflow Jobs to run at intervals that match how frequently your external schema changes. This proactive approach is useful immediately after creating a foreign catalog, when the first query would otherwise trigger a full metadata refresh.

Grant access and query foreign data

Foreign catalogs appear in Unity Catalog alongside your standard catalogs, making them subject to the same permission model. You control access by granting appropriate privileges to users and groups.

Grant read privileges to users who need to query foreign data:

```
GRANT USE CATALOG ON CATALOG prod_customer_data TO `data-analysts`;
GRANT USE SCHEMA ON SCHEMA prod_customer_data.public TO `data-analysts`;
GRANT SELECT ON TABLE prod_customer_data.public.transactions TO `data-analysts`;
```

These grants follow Unity Catalog's hierarchical permission model. Users need `USE CATALOG` on the catalog, `USE SCHEMA` on the schema, and `SELECT` on specific tables. This granular control lets you expose some foreign tables while restricting others, even though they all exist in the same external database.

Query foreign tables using standard SQL syntax. Once permissions are in place, users query foreign data exactly as they would query native Unity Catalog tables:

```
SELECT
  customer_id,
  SUM(transaction_amount) AS total_spent
FROM prod_customer_data.public.transactions
WHERE transaction_date >= '2024-01-01'
GROUP BY customer_id
ORDER BY total_spent DESC
LIMIT 100;
```

Azure Databricks translates this query into SQL appropriate for PostgreSQL, pushes the query down to the external database, and returns results. The external database performs the aggregation and filtering, leveraging its own compute resources while Azure Databricks handles the final result set.

Query execution uses remote compute in the external system. Understanding this behavior helps you optimize performance. Complex joins, aggregations, and filters are pushed down to the external database when possible, reducing data transfer. However, if your result set is very large, the Databricks executor retrieving data might run out of memory. Monitor query performance and consider using materialized views to cache frequently accessed external data locally.

9. Configure AI/BI Genie instructions

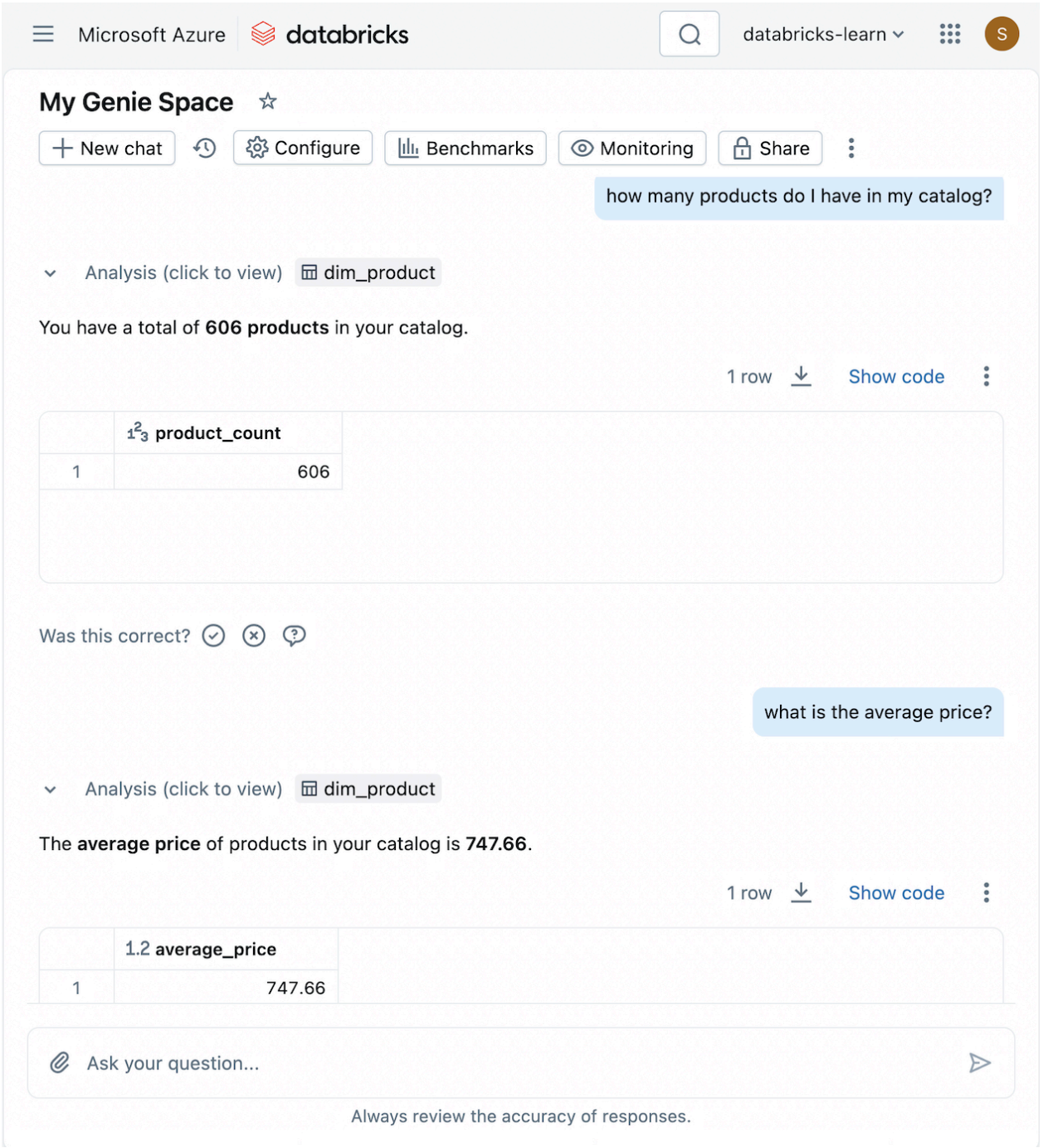
https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/9-configure-ai-bi-genie-instructions

Configure AI/BI Genie instructions

Completed

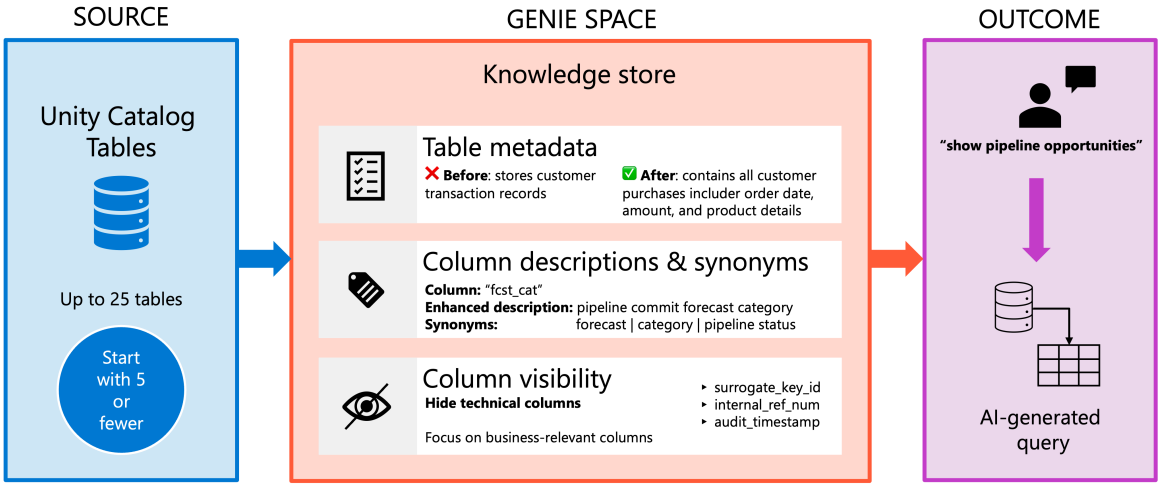
Making your data discoverable involves more than organizing it into catalogs and schemas. **AI/BI Genie** in Azure Databricks enables business users to explore data through **natural language questions**, but it needs guidance to interpret your organization's terminology and business logic.

When you configure AI/BI Genie instructions properly, you transform Unity Catalog metadata into a **knowledge store** that helps both humans and AI systems understand your data.



Configure the knowledge store

The **knowledge store** is your primary tool for teaching AI/BI Genie about your data. You can customize **metadata** at the **space level** without altering the underlying Unity Catalog objects. This gives you flexibility to refine descriptions and add context specific to how business users interact with the data.



When you create a Genie space, you start by selecting up to **25 tables or views** from Unity Catalog. These data objects form the foundation of your space. The key is choosing tables that answer a focused set of business questions rather than including everything available. A sales manager might need access to opportunity and account tables, while a logistics manager requires shipment and inventory data. Keep your initial selection small. **Five tables or fewer** works well. Expand only as users request additional data.

After adding tables, you enhance their discoverability by editing metadata. Select **Configure > Data** in your Genie space, then select a table to view its columns. Each table includes a **description field** where you can explain its purpose using **business language**. Instead of technical descriptions like "stores customer transaction records," write "contains all customer purchases including order date, amount, and product details." This context helps Genie generate accurate SQL when users ask questions about purchases or orders.

Column-level metadata requires similar attention. Unity Catalog preserves column names and descriptions from the source tables, but you can refine them within the Genie space. Suppose you have a column named `fcst_cat` that stores forecast categories. In the space, you update its description to "indicates whether an opportunity is in the Pipeline, Best Case, or Commit forecast category" and add **synonyms** like "forecast," "category," and "pipeline status." When users ask about pipeline opportunities, Genie matches their language to your column using these synonyms.

You can also **hide columns** that don't serve business users. Technical columns like surrogate keys, internal identifiers, or audit timestamps often confuse both users and AI. Select the columns you want to hide and choose **Hide selected columns** from the **Actions** menu. This keeps Genie focused on columns that matter for business analysis.

Define data relationships and prompt matching

With metadata in place, you help Genie understand how your tables connect and what values they contain. These configurations allow Genie to write accurate **JOIN statements** and match user questions to specific data values.

Unity Catalog can store **foreign key relationships** between tables, and Genie uses these to construct `JOIN` statements automatically. If your tables don't have foreign keys defined in Unity Catalog, or if you need more complex join logic, define relationships in the knowledge store. Navigate to the **Joins** section and specify how tables relate to each other.

For a sales pipeline dataset, you define a join between an `opportunity` table and an `accounts` table. Select both tables, enter the join condition `opportunity.accountid = accounts.id`, and specify the relationship type as **Many to one** because multiple opportunities can belong to a single account. This teaches Genie how to combine these tables when users ask questions like "show me opportunities by account region."

Prompt matching complements relationship definitions by giving Genie visibility into your actual data. When enabled, Genie collects representative values from columns, helping it match user prompts to real values in your tables. If a user asks about "opportunities in California" but your data uses state codes like "CA," prompt matching allows Genie to make that connection. Two components control how matching works: **format assistance** provides representative values for all eligible columns so Genie understands data types and formatting patterns, while **entity matching** curates distinct value lists for categorical columns. The system automatically enables prompt matching when you add tables, but you can manage which columns participate.

Entity matching is most useful for categorical columns where users are likely to reference specific entries—such as states, product categories, and region names. Genie can store up to **1,024 distinct values** per column across **120 columns** in your space. To enable entity matching for a column, select the pencil icon next to the column name, expand **Advanced settings**, and turn on **Entity matching**. As your data changes over time, refresh prompt matching data to keep the knowledge store current.

Add SQL instructions and examples

Beyond metadata and relationships, you teach Genie how to answer business questions by providing **SQL instructions**. Two types of instructions serve different purposes: **SQL expressions** define reusable business concepts, while **example SQL queries** show how to handle specific question patterns.

SQL expressions work best for common business terms that appear across many questions. Navigate to **Configure > Instructions > SQL Expressions** to define measures, filters, and dimensions. A measure might calculate gross margin as `(revenue - cost) / revenue`, while a dimension could extract the fiscal quarter from a date field. When you define these expressions with clear names and synonyms, Genie can incorporate them into generated queries without you writing complete SQL examples.

Example SQL queries address more complex scenarios where users ask multi-part questions or use domain-specific phrasing. These queries appear in the **SQL Queries** tab under **Instructions**. For each example, you write both the question users might ask and the SQL that answers it. If users frequently ask "What's our current pipeline by region?", provide that exact phrasing along with SQL that joins opportunity and account tables, filters for open opportunities in the pipeline forecast category, and groups results by region.

```
-- What's our current pipeline by region?
SELECT
  accounts.region AS Region,
  SUM(opportunity.amount) AS Pipeline
FROM opportunity
JOIN accounts ON opportunity.accountid = accounts.id
WHERE
  opportunity.forecastcategory = 'Pipeline'
  AND opportunity.stagename NOT LIKE '%closed%'
GROUP BY accounts.region;
```

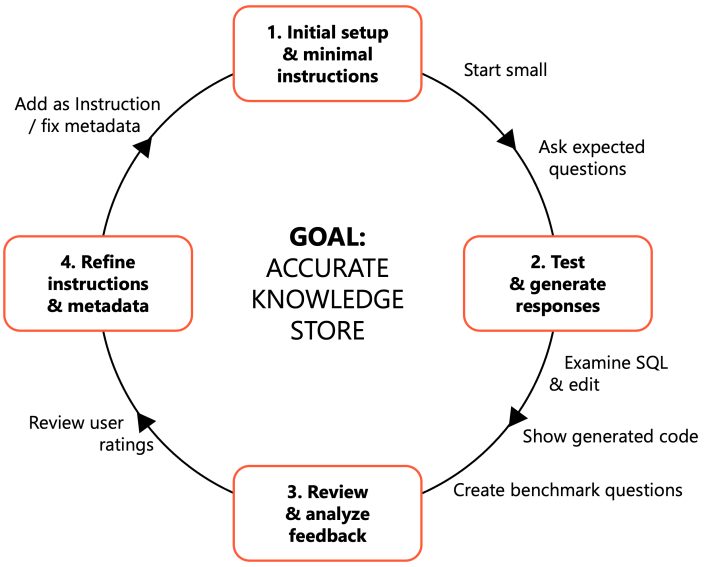
Genie can either use example queries directly or learn from them to answer similar questions. When you add **parameters** to example queries using the syntax `:parameter_name`, Genie substitutes user-provided values at runtime. Parameterized example queries that match the user's exact question are marked as **Trusted** to give users confidence in the results. Non-parameterized example queries are used as context to guide Genie in generating similar SQL, but aren't marked as Trusted.

The **Text** tab holds general instructions that apply globally across all questions. Use this sparingly for context that doesn't fit into other formats. Examples include explaining that your fiscal year starts in February or that all timestamps should be converted to a specific timezone. Too many general instructions can dilute their effectiveness, so focus on essential business rules that Genie can't infer from your data.

You can add up to **100 instructions** total across SQL queries, SQL functions, and the general text instructions block. Each example SQL query counts as one, each SQL function counts as one, and the entire general instructions text block counts as one. SQL expressions have a separate limit of **200** and do not count toward the 100-instruction total. These limits encourage you to prioritize quality over quantity.

Apply iterative refinement

Configuring AI/BI Genie is an **ongoing process**, not a one-time setup. Start with minimal instructions and expand based on actual user questions and feedback.



After configuring your initial space, test it yourself by asking questions you expect business users to pose. Review the generated SQL carefully to identify where Genie misinterprets data or business terminology.

When you find incorrect responses, use the **Show generated code** option to examine the SQL, edit it as needed, and save it as a new instruction by clicking **Add as instruction**. This teaches Genie how to handle similar questions in the future. You can also create **benchmark questions** that test Genie's accuracy systematically. Add common questions with their expected SQL answers, then run benchmarks regularly as you refine the space to measure improvement.

Encourage business users to provide feedback through the built-in response rating mechanism. When users mark responses as correct or incorrect, you can review their feedback in the **Monitoring** tab and adjust instructions accordingly. This collaborative refinement ensures your Genie space evolves to meet real business needs rather than assumptions about how users might ask questions.

As you iterate, watch for patterns in failed queries. If Genie consistently uses the wrong table or column, review your metadata and descriptions. If it can't match user language to data values, check that prompt matching is enabled for relevant columns. If joins are incorrect, verify your relationship definitions. Each refinement makes the knowledge store more accurate and the space more valuable to users.

10. Exercise - Create and Organize Objects in Unity Catalog

https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/10-exercise-create-organize-objects-unity-catalog

Exercise - Create and Organize Objects in Unity Catalog

Completed

Now it's your chance to create and organize objects in Unity Catalog. In this lab, you build a complete Unity Catalog namespace for a university data platform — creating a catalog, medallion schemas, managed tables with primary and foreign key constraints, views, a volume, and a reusable SQL function. You practice DDL operations such as adding columns and applying governance tags, and explore how Unity Catalog organizes and governs structured data at every layer of the medallion architecture. By the end, you'll have a fully structured, query-ready environment that reflects real-world data engineering practices on Azure Databricks.

> **Note**
>
> To complete this lab, you need an Azure subscription in which you have administrative access.

Launch the exercise and follow the instructions.

Launch Exercise

11. Knowledge check

https://learn.microsoft.com/en-us/training/modules/create-and-organize-objects-in-unity-catalog/11-knowledge-check

Knowledge check

Completed

12. Summary

Summary

Completed

You've learned how to create and organize Unity Catalog objects to build a comprehensive data governance framework. Starting with catalogs for environment isolation, you progressed through schemas for logical organization, then created tables, views, and volumes for storing and accessing data. You explored advanced capabilities like foreign catalogs for external database integration and AI/BI Genie instructions for natural language data discovery.

The three-layer namespace structure—catalogs, schemas, and objects—provides the flexibility to organize data according to your organization's needs while maintaining centralized security controls. Managed tables and volumes simplify data lifecycle management, while external objects let you govern existing data without migration. Dynamic views enable row-level and column-level security, and materialized views improve performance for complex analytics queries.

Effective naming conventions and well-configured metadata make your data discoverable and maintainable over time. As your organization's data platform grows, these foundational organizational patterns scale to accommodate new teams, projects, and use cases while preserving the governance boundaries you've established.

Apply these concepts in your own Azure Databricks environment by starting small—create a catalog for development work, add a schema for your current project, and experiment with different table and view types. As you gain confidence, expand your catalog structure and refine your naming conventions to match your organization's governance requirements.