# Azure Databricks

**Link:** <https://portal.azure.com/#view/HubsExtension/BrowseResource/resourceType/Microsoft.Databricks%2Fworkspaces>

Introduction:

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 deeply with other services provided by Microsoft Azure, offering a seamless experience for data preparation, building machine learning models, and data analysis.

Key features of Azure Databricks include its native integration with **Microsoft Entra ID**, and its capability to use other Azure services such as **Azure Storage, Azure Data Lake Storage, and Azure Cosmos DB**. The platform also offers an interactive workspace that facilitates collaboration among data scientists, data engineers, and business analysts. This collaborative environment supports various programming languages like **Python, Scala, R, and SQL**, allowing teams to develop and iterate on their data models efficiently. Moreover, Azure Databricks is designed to scale easily, managing both the computational demands of machine learning algorithms and the processing needs of large data sets.

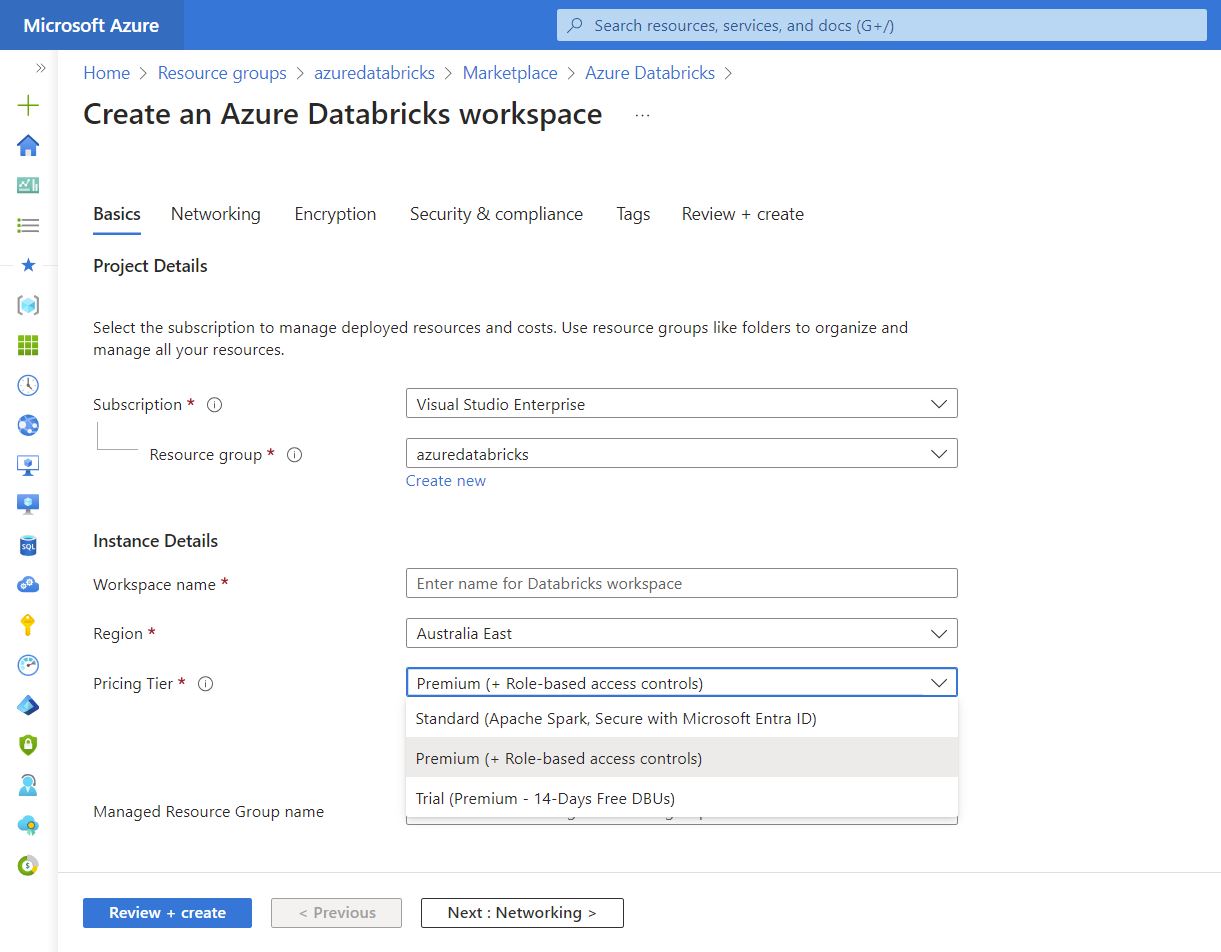
Creating an Azure Databricks workspace:

To use Azure Databricks, you must create an Azure Databricks workspace in your Azure subscription, by:

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

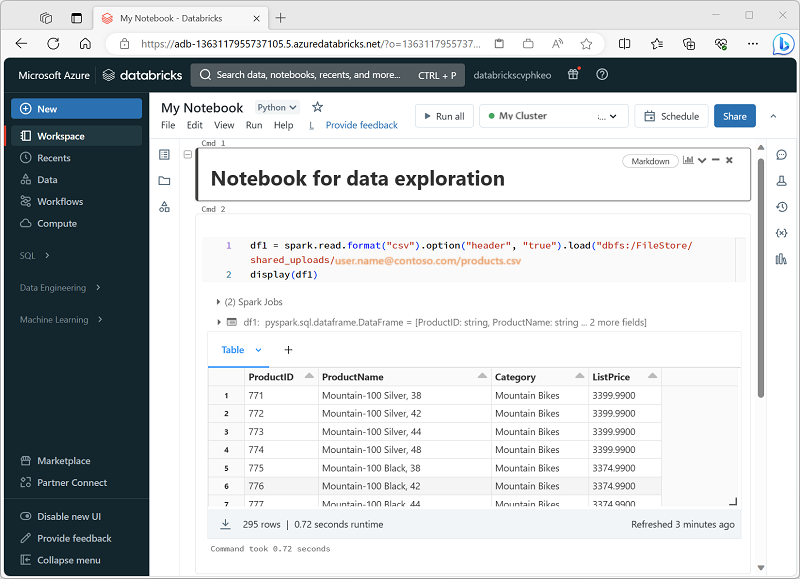
When you create a workspace, you must specify one of the following pricing tiers.

* **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



Using the Azure Databricks Portal:

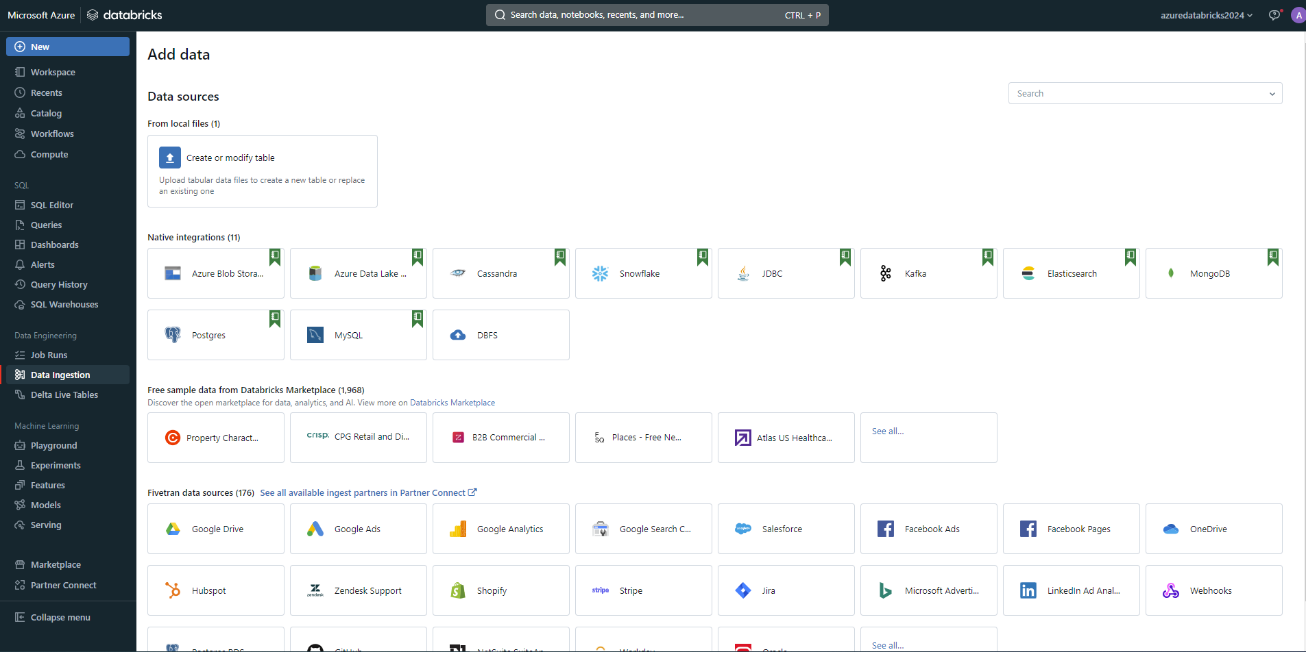
After you provision an Azure Databricks workspace, you can use the **Azure Databricks portal** to work with data and compute resources. The Azure Databricks portal is a web-based user interface where you can create and manage workspace resources, such as Spark clusters, and use notebooks and queries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



Identify Azure Databricks workloads:

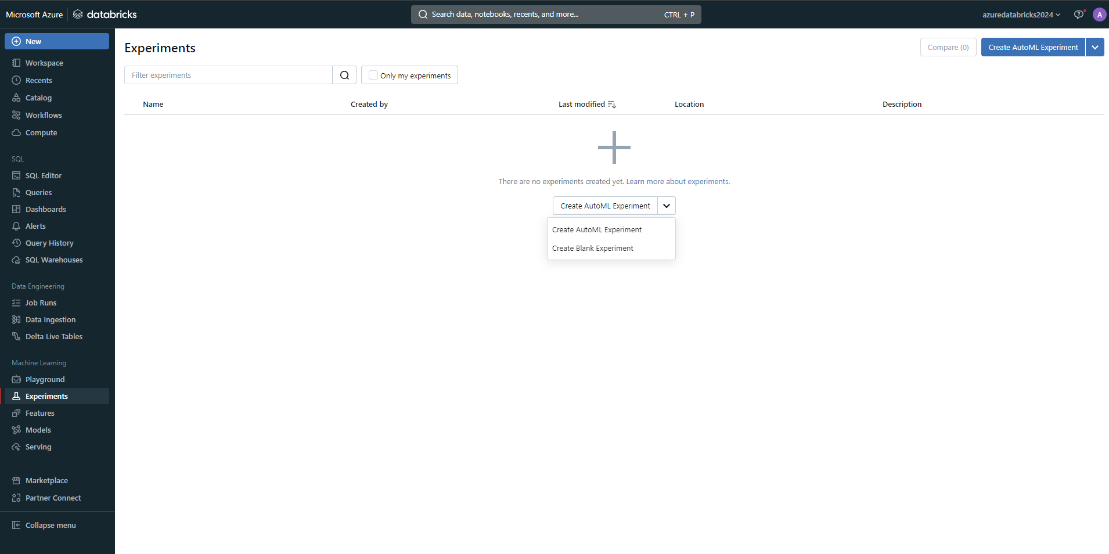
1. **Data Science and Engineering**

This workload is designed 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.



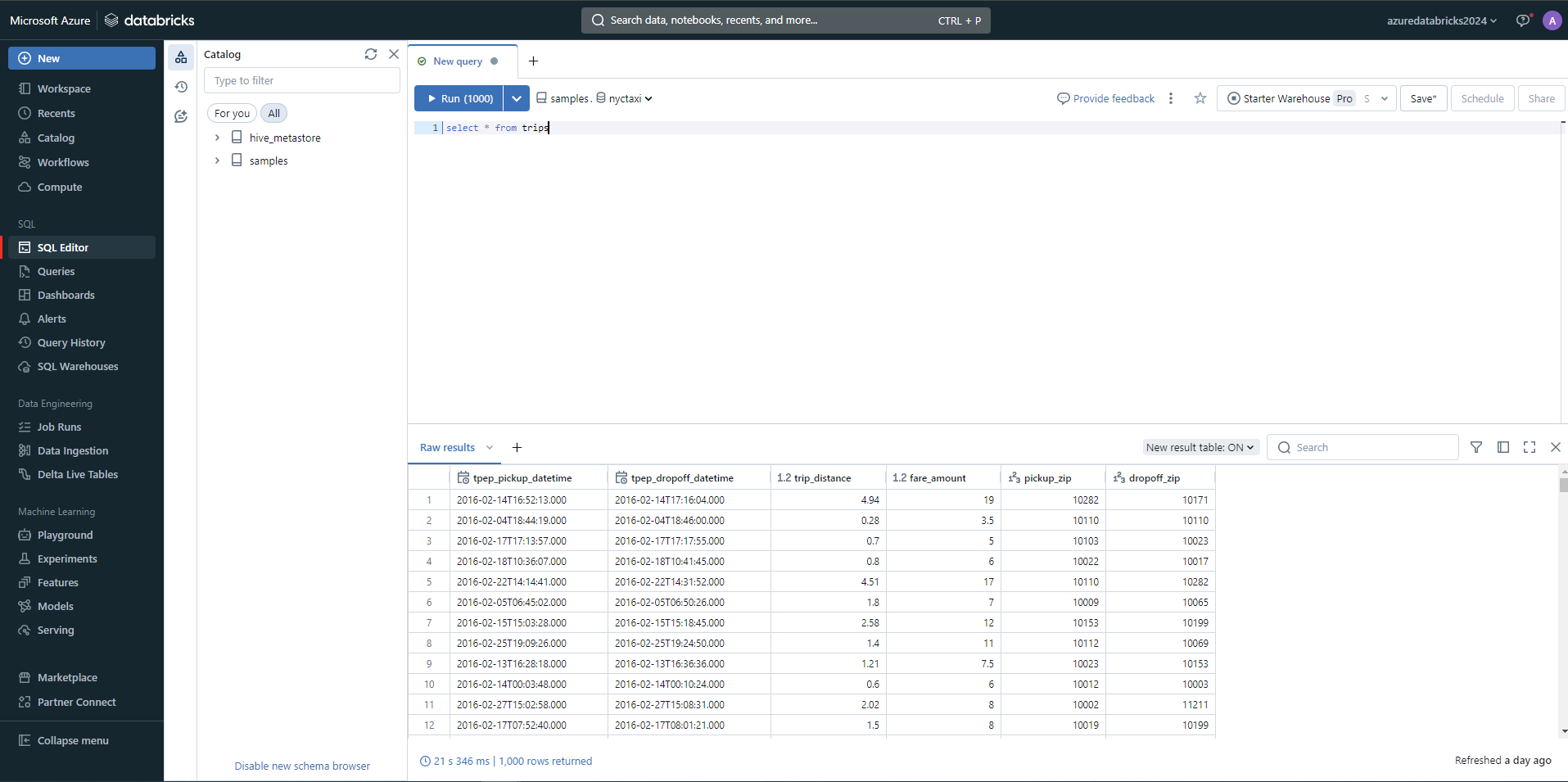
1. **Machine Learning**

The Machine Learning workload on Azure Databricks is optimized for 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.



1. **SQL**

The SQL workload is geared towards data analysts who primarily interact with data through SQL. It 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.



Key Concepts:

1. **Workspaces**

A workspace is an environment for accessing all the Databricks assets. It provides a user interface to manage notebooks, libraries, and experiments. Workspaces can be organized into folders and shared among team members, facilitating collaboration and resource management.

1. **Notebooks**

Databricks notebooks are interactive documents that contain runnable code, visualizations, and narrative text. They support multiple languages, including **Python, R, Scala, and SQL**, which can be used simultaneously within the same notebook. Notebooks are central to collaborative projects and are ideal for exploratory data analysis, data visualization, and complex data workflows.

1. **Clusters**

Clusters are the computational engines of Azure Databricks. Users can create and scale clusters according to the computational resources needed. Clusters can be configured manually or set to auto-scale based on workload. They support different types of nodes for various tasks, like driver and worker nodes, ensuring efficient resource utilization.

1. **Jobs**

Jobs in Azure Databricks are used to schedule and run automated tasks. These tasks can be notebook runs, Spark jobs, or arbitrary code executions. Jobs can be triggered on a schedule or run in response to certain events, making it easy to automate workflows and periodic data processing tasks.

1. **Databricks Runtime**

The Databricks Runtime is a set of performance-optimized versions of **Apache Spark**. It includes enhancements for improved performance and additional functionality beyond standard Spark, such as optimizations for machine learning workloads, graph processing, and genomics.

1. **Delta Lake**

Delta Lake is an open-source storage layer that brings reliability and scalability to data lakes. It provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing, all crucial for managing large-scale data in a consistent and fault-tolerant manner.

1. **MLflow**

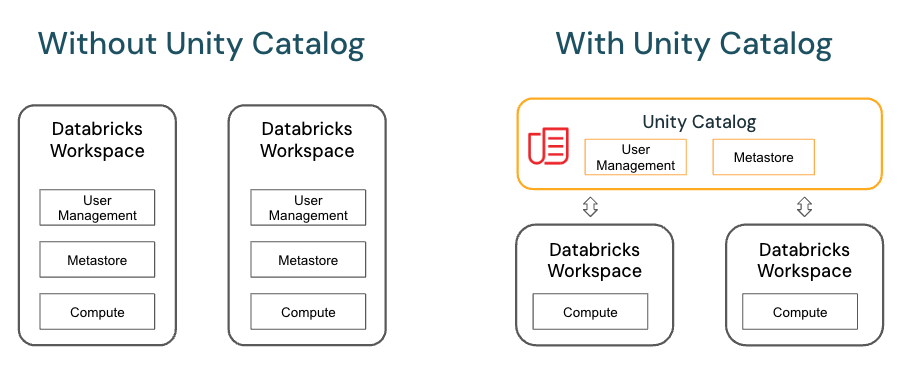
MLflow is an open-source platform for managing the end-to-end machine learning lifecycle. It includes features for experiment tracking, model management, and deployment, helping practitioners manage and share their ML models and experiments efficiently.

Data governance:

Data governance is critical for ensuring that data within an organization is managed securely, efficiently, and in compliance with regulations. Azure Databricks, combined with **Unity Catalog** and **Microsoft Purview**, provides a robust solution for managing and governing data effectively.

1. **Unity Catalog**

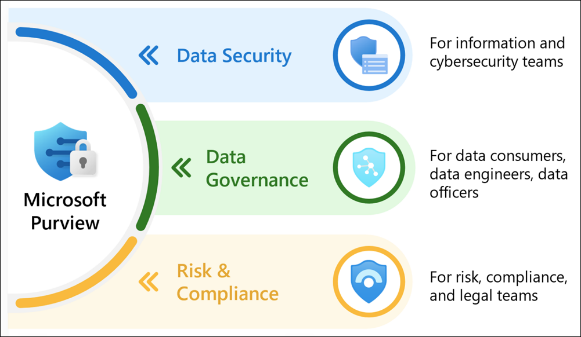
Unity Catalog in Azure Databricks is a centralized metastore that manages metadata for all data assets and AI assets across Databricks workspaces. It enables fine-grained security controls and governance policies at scale, making it easier to manage data across multiple teams and projects securely.



* **Unified Namespace**: Unity Catalog offers a single namespace for datasets, files, and machine learning models across all workspaces, making it easier to manage and discover assets.
* **Fine-grained Access Control**: It allows administrators to set precise access controls on data using standard **SQL GRANT** and **REVOKE** statements, aligning with the principle of least privilege.
* **Data Lineage**: Unity Catalog captures and displays data lineage, which is critical for tracking the flow of data and understanding its transformations over time.
* **Centralized Metadata Management**: Manages all metadata centrally, ensuring that definitions, descriptions, and other metadata are consistent across projects and workspaces.
* **Integration with Databricks SQL**: Unity Catalog is fully integrated with **Databricks SQL**, allowing for seamless querying and management of data assets without moving data out of the platform.

1. **Microsoft Purview**

Microsoft Purview offers a suite of data governance tools designed to provide visibility, control, and insights into data usage across an organization. It helps you discover, classify, protect, and monitor data, no matter where it resides.



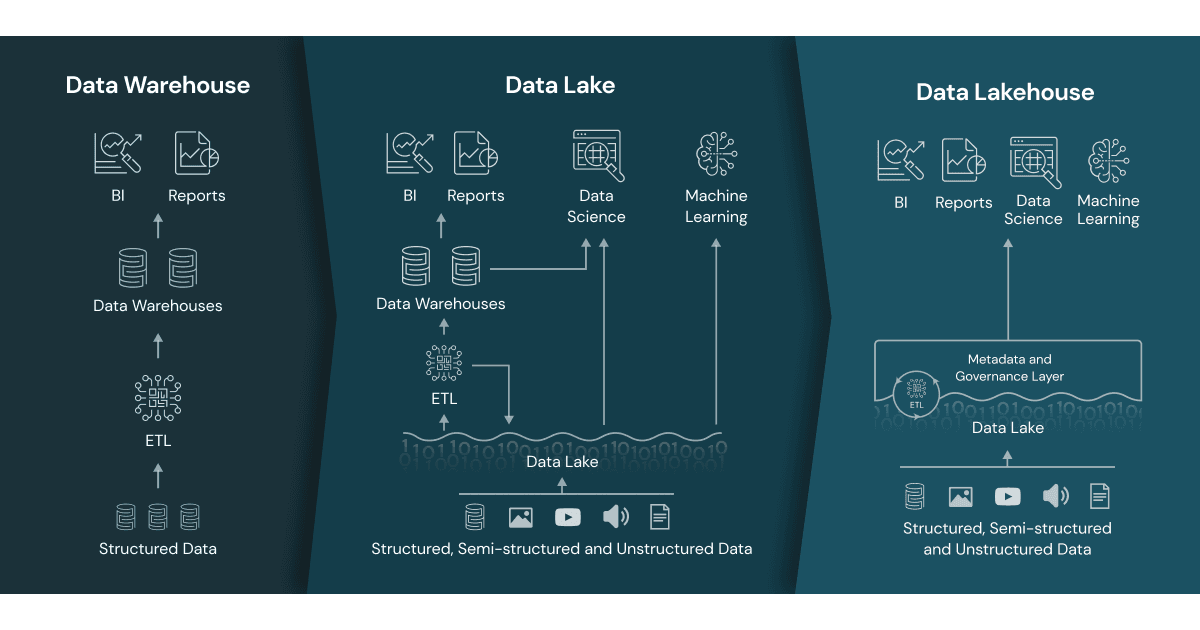
* **Data Discovery and Classification**: Automatically scan and classify data across your Azure Databricks environment using built-in classifiers and also create custom classifiers.
* **Data Lineage**: Provides detailed visibility into the data lineage, showing how data is transformed and moved across different systems and processes, including within Azure Databricks.
* **Data Map**: Aggregates metadata from various data sources into a searchable catalog, allowing users to understand the data landscape and its relationships.
* **Access and Policy Management**: Allows for the creation of governance policies that enforce how data is accessed and used within Azure Databricks and other integrated systems.
* **Insights and Reporting**: Offers detailed reports on data discovery, sensitivity classification, and access analytics, helping to ensure compliance and optimize data governance strategies.

# Data Lakehouse Architecture

The **Databricks Data Intelligence Platform** is built on lakehouse architecture, which combines the best elements of data lakes and data warehouses to reduce costs and deliver on data and AI initiatives faster.

1. **Unified:**

* One architecture for integration, storage, processing, governance, sharing, analytics and AI. One approach to how you work with structured and unstructured data. One end-to-end view of data lineage and provenance. One toolset for Python and SQL, notebooks and IDEs, batch and streaming, and all major cloud providers.



1. **Open:**

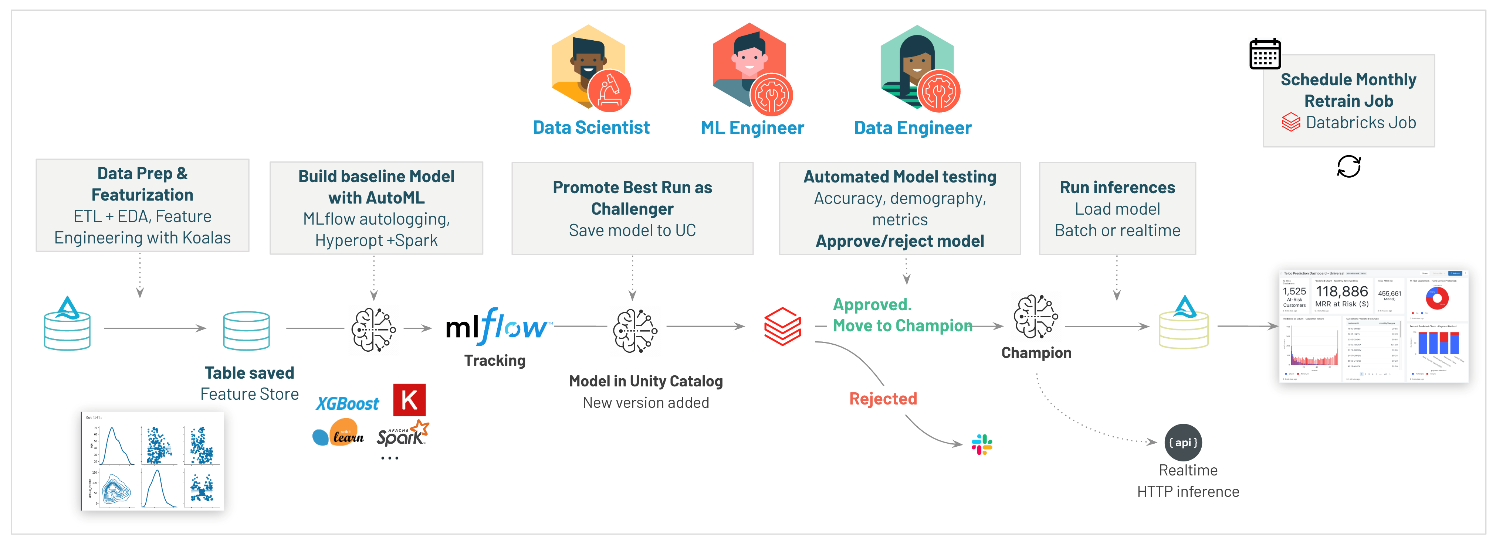
* With Databricks, data is always under your control, free from proprietary formats and closed ecosystems.
* Lakehouse is underpinned by widely adopted open source projects Apache Spark, Delta Lake and MLflow, and is globally supported by the Databricks Partner Network. ​​And Delta Sharing provides an open solution to securely share live data from your lakehouse to any computing platform without replication and complicated ETL.

1. **Scalable:**

* Automatic optimization for performance and storage ensures the lowest TCO of any data platform together with world-record-setting performance for both data warehousing and AI use cases — including generative techniques like large language models (LLMs).
* Whether you’re a startup or global enterprise, Databricks is built to meet the demands of business at any scale.

# Machine Learning Operation (MLOps)

Tutorial: <https://www.databricks.com/resources/demos/tutorials/data-science-and-ai/mlops-end-to-end-pipeline?itm_data=demo_center>



What is MLOps

MLOps is a set of processes and automated steps for managing code, data, and models to improve performance, stability, and long-term efficiency of ML systems. It combines **DevOps**, **DataOps**, and **ModelOps.**



ML assets such as code, data, and models are developed in stages that progress from early development stages that do not have tight access limitations and are not rigorously tested, through an intermediate testing stage, to a final production stage that is tightly controlled. The Databricks platform manages these assets on a single platform with unified access control. You can develop data applications and ML applications on the same platform, reducing the risks and delays associated with moving data around.

General recommendations for MLOps

1. **Create a separate environment for each stage**:

* An execution environment is the place where models and data are created or consumed by code. Each execution environment consists of compute instances, their runtimes and libraries, and automated jobs.
* Creating separate environments for the different stages of ML code and model development with clearly defined transitions between stages:
* Development
* Staging
* Production

1. **Access control and versioning:**

* **Use Git for version control**. Pipelines and code should be stored in Git for version control. Moving ML logic between stages can then be interpreted as moving code from the development branch, to the staging branch, to the release branch. Use **Databricks Git folders** to integrate with Git provider and sync notebooks and source code with **Databricks workspaces**. [Git integration with Databricks Git folders | Databricks on AWS](https://docs.databricks.com/en/repos/index.html)
* **Store data in a lakehouse architecture using Delta tables**. Data should be stored in a lakehouse architecture in your cloud account. Both raw data and feature tables should be stored as Delta tables with access controls to determine who can read and modify them.
* **Manage model development with MLflow**. Use MLflow to track the model development process and save code snapshots, model parameters, metrics, and other metadata.
* **Use Models in Unity Catalog to manage the model lifecycle**. Use Models in Unity Catalog to manage model versioning, governance, and deployment status.

1. **Deploy code, not models:**

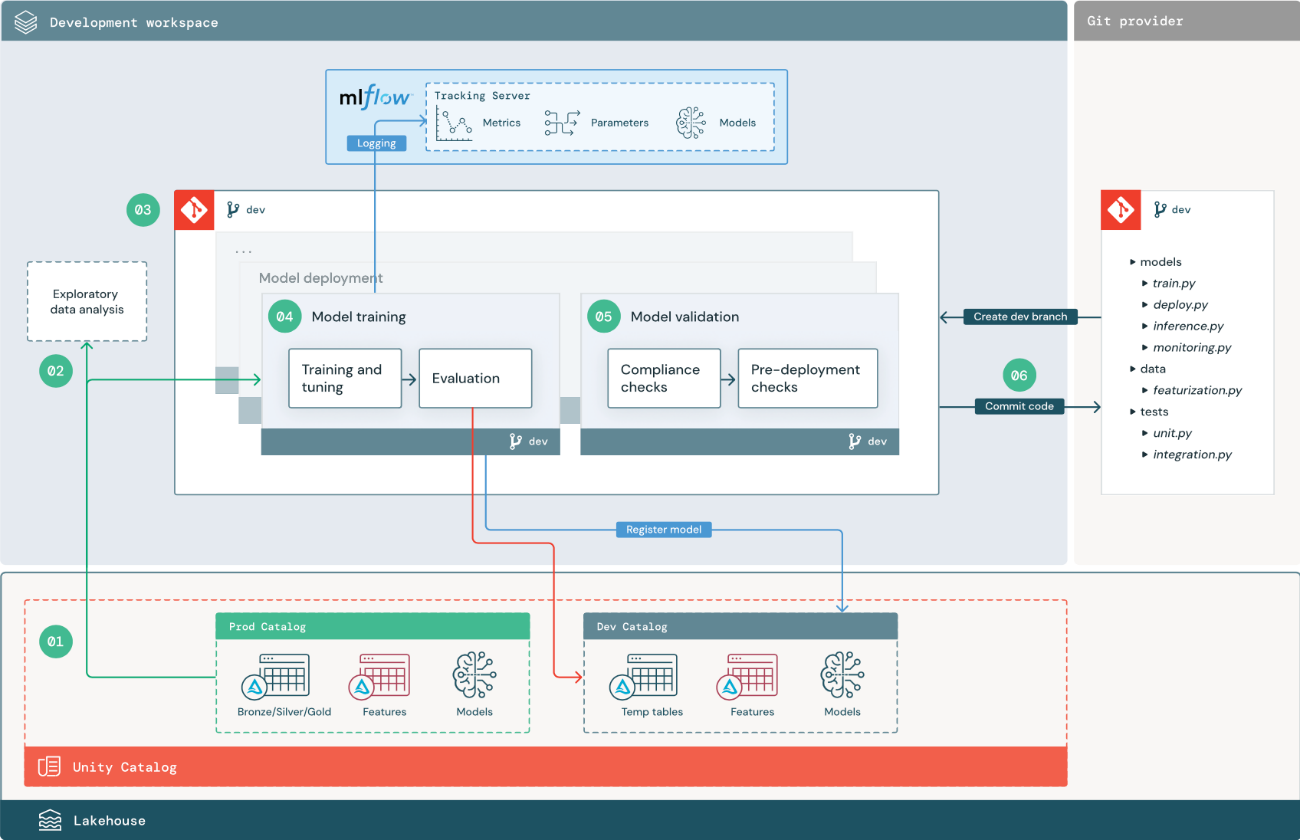
* In most situations, Databricks recommends that during the ML development process, you promote code, rather than models, from one environment to the next. Moving project assets this way ensures that all code in the ML development process goes through the same code review and integration testing processes. It also ensures that the production version of the model is trained on production code. [Model deployment patterns | Databricks on AWS](https://docs.databricks.com/en/machine-learning/mlops/deployment-patterns.html)

# Recommended MLOps workflow

The following sections describe a typical MLOps workflow, covering each of the three stages: **development, staging, and production**. This section uses the terms “data scientist” and “ML engineer” as archetypal personas; specific roles and responsibilities in the MLOps.

Development stage

The focus of the development stage is experimentation. **Data scientists** develop features and models and run experiments to optimize model performance. The output of the development process is ML pipeline code that can include feature computation, model training, inference, and monitoring.



1. **Data sources:**

* The development environment is represented by the **dev catalog** in **Unity Catalog**. Data scientists have read-write access to the dev catalog as they create temporary data and feature tables in the development workspace. Models created in the development stage are registered to the dev catalog.
* Ideally, data scientists working in the development workspace also have read-only access to production data in the **prod catalog**. Allowing data scientists read access to production data, inference tables, and metric tables in the prod catalog enables them to analyze current production model predictions and performance. Data scientists should also be able to load production models for experimentation and analysis.

1. **Exploratory data analysis (EDA):**

* Data scientists explore and analyze data in an interactive, iterative process using **notebooks**. The goal is to assess whether the available data has the potential to solve the business problem. In this step, the data scientist begins identifying data preparation and featurization steps for model training. This ad hoc process is generally not part of a pipeline that will be deployed in other execution environments.
* **Databricks AutoML** accelerates this process by generating baseline models for a dataset. AutoML performs and records a set of trials and provides a Python notebook with the source code for each trial run, so you can review, reproduce, and modify the code. AutoML also calculates summary statistics on your dataset and saves this information in a notebook.

1. **Code:**

* The code repository contains all of the pipelines, modules, and other project files for an ML project. Data scientists create new or updated pipelines in a **development (“dev”) branch** of the project repository. Starting from EDA and the initial phases of a project, data scientists should work in a repository to share code and track changes.

1. **Train model (development):**

* Data scientists develop the model training pipeline in the development environment using tables from the dev or prod catalogs. This pipeline includes **2 tasks**:
  + **Training and tuning**: The training process logs model parameters, metrics, and artifacts to the **MLflow Tracking server**. After training and tuning hyperparameters, the final model artifact is logged to the tracking server to record a link between the model, the input data it was trained on, and the code used to generate it.
  + **Evaluation:** Evaluate model quality by testing on held-out data. The results of these tests are logged to the **MLflow Tracking server**. The purpose of evaluation is to determine if the newly developed model performs better than the current production model.
* If your organization’s governance requirements include additional information about the model, you can save it using **MLflow tracking**. Typical artifacts are plain text descriptions and model interpretations such as plots produced by **SHAP (Game theoretic approach to explain the output of any machine learning model)**. Specific governance requirements may come from a data governance officer or business stakeholders.
* The output of the model training pipeline is an ML model artifact stored in the **MLflow Tracking server** for the development environment.
* When the model training is complete, register the model to **Unity Catalog**. Set up your pipeline code to register the model to the catalog corresponding to the environment that the model pipeline was executed in.
* With the recommended architecture, you deploy a **multitask Databricks workflow** in which the first task is the model training pipeline, followed by model validation and model deployment tasks. The model training task yields a model URI (Uniform Resource Identifier) that the model validation task can use. You can use task values to pass this URI to the model.

1. **Validate and deploy model (development):**

* **Model validation.** The model validation pipeline takes the model URI from the model training pipeline, loads the model from **Unity Catalog**, and runs validation checks.
  + **Validation checks** depend on the context. They can include fundamental checks such as confirming format and required metadata, and more complex checks that might be required for highly regulated industries, such as predefined compliance checks and confirming model performance on selected data slices.
  + The primary function of the model validation pipeline is to determine whether a model should proceed to the deployment step. If the model passes pre-deployment checks, it can be assigned the “**Challenger**” alias in **Unity Catalog**. If the checks fail, the process ends. You can configure your workflow to notify users of a validation failure. [Add email and system notifications for job events | Databricks on AWS](https://docs.databricks.com/en/jobs/job-notifications.html)
* **Model deployment.** The model deployment pipeline typically either directly promotes the newly trained **“Challenger”** model to **“Champion”** status using an alias update or facilitates a comparison between the existing **“Champion”** model and the new **“Challenger”** model. This pipeline can also set up any required inference infrastructure, such as **Model Serving endpoints**.

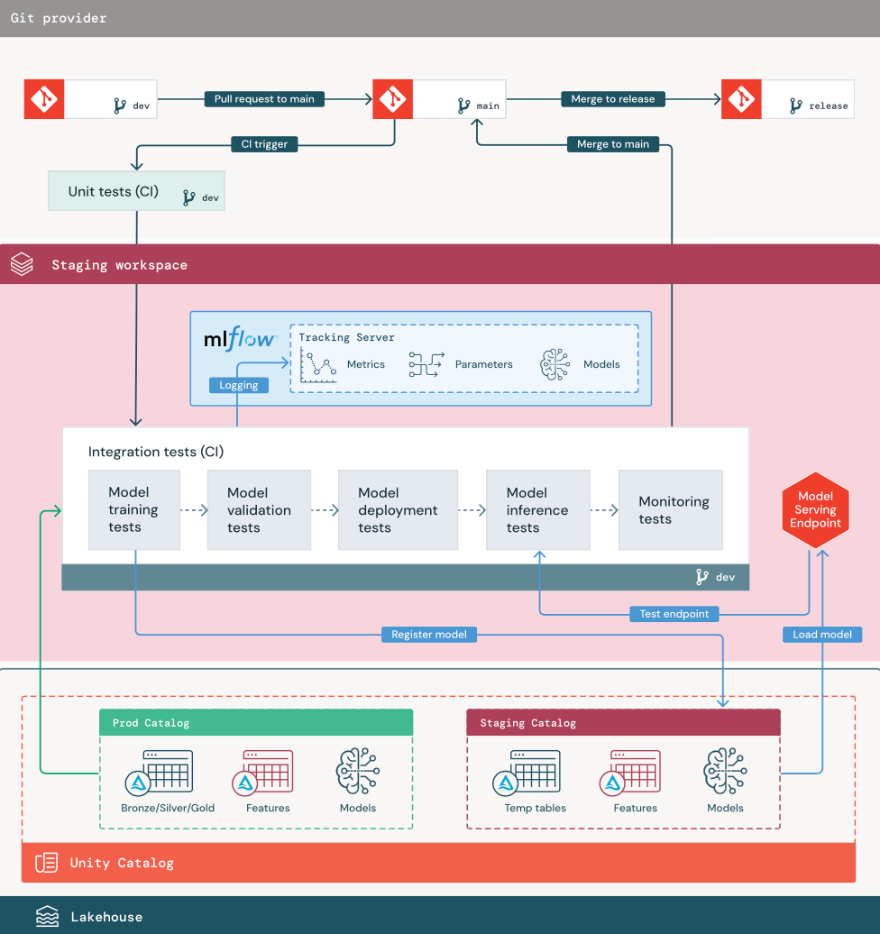
1. **Commit code:**

* After developing code for training, validation, deployment and other pipelines, the data scientist or ML engineer commits the dev branch changes into source control.

Staging stage

The focus of this stage is testing the ML pipeline code to ensure it is ready for production. All of the ML pipeline code is tested in this stage, including code for model training as well as feature engineering pipelines, inference code, and so on.

**ML engineers** create a **CI pipeline** to implement the unit and integration tests run in this stage. The output of the staging process is a release branch that triggers the CI/CD system to start the production stage.



1. **Data:**

* The staging environment should have its own catalog in **Unity Catalog** for testing ML pipelines and registering models to Unity Catalog. Assets written to **“staging” catalog** are generally temporary and only retained until testing is complete. The development environment may also require access to the staging catalog for debugging purposes.

1. **Merge code:**

* **Data scientists** develop the model training pipeline in the development environment using tables from the development or production catalogs.
  + **Pull request**. The deployment process begins when a pull request is created against the main branch of the project in source control **(Git provider).**
  + **Unit tests (CI)**. The pull request automatically builds source code and triggers unit tests. If unit tests fail, the pull request is rejected. **Unit tests** are part of the software development process and are continuously executed and added to the codebase during the development of any code. Running unit tests as part of a CI pipeline ensures that changes made in a development branch do not break existing functionality.

1. **Integration tests (CI):**

* The CI process then runs the integration tests. Integration tests run all pipelines (including feature engineering, model training, inference, and monitoring) to ensure that they function correctly together. The staging environment should match the production environment as closely as is reasonable.
* If you are deploying an ML application with real-time inference, you should create and test serving infrastructure in the staging environment. This involves triggering the model deployment pipeline, which creates a serving endpoint in the staging environment and loads a model.
* To reduce the time required to run integration tests, some steps can trade off between fidelity of testing and speed or cost. For example, if models are expensive or time-consuming to train, you might use small subsets of data or run fewer training iterations. For model serving, depending on production requirements, you might do full-scale load testing in integration tests, or you might just test small batch jobs or requests to a temporary endpoint.

1. **Merge to staging branch:**

* If all tests pass, the new code is merged into the main branch of the project. If tests fail, the CI/CD system should notify users and post results on the pull request.
* You can schedule periodic integration tests on the main branch. This is a good idea if the branch is updated frequently with concurrent pull requests from multiple users.

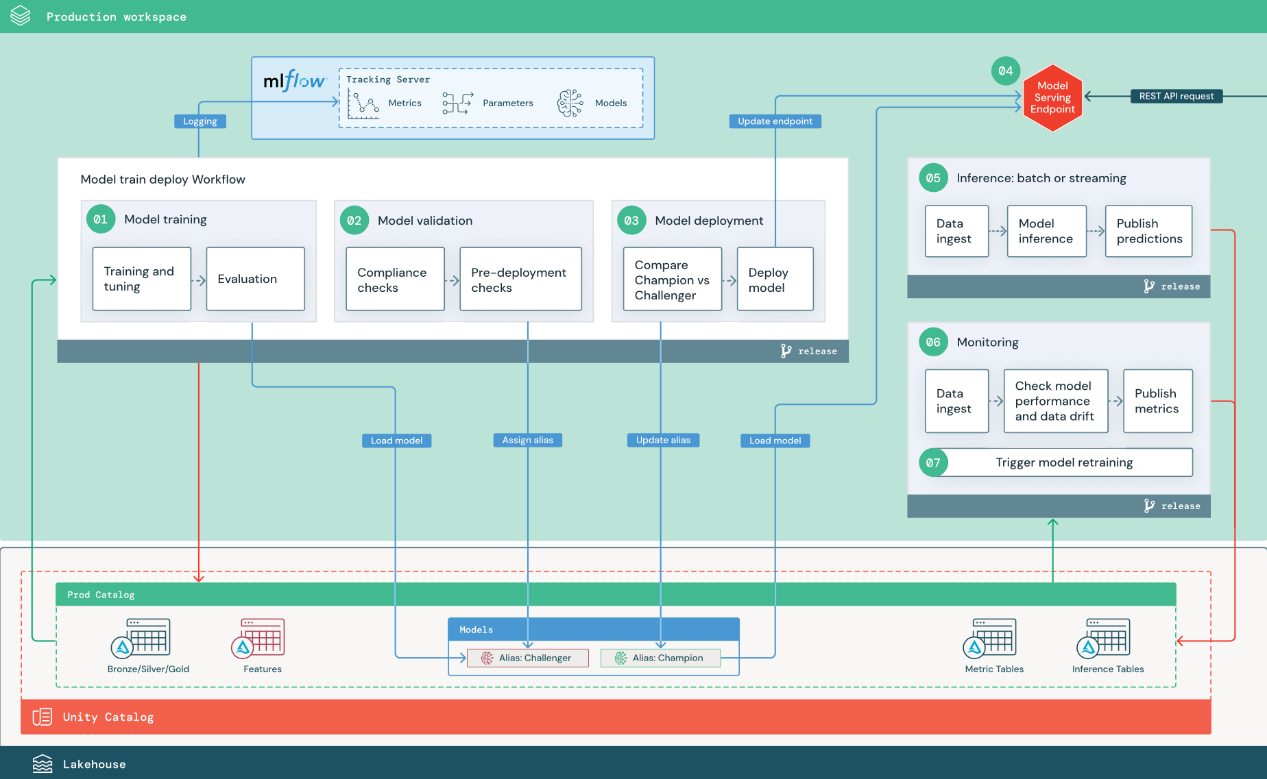
5. **Create a release branch:**

* After CI tests have passed and the dev branch is merged into the main branch, the **ML engineer** creates a release branch, which triggers the CI/CD system to update production jobs.

Production stage

**ML engineers** own the production environment where ML pipelines are deployed and executed. These pipelines trigger model training, validate and deploy new model versions, publish predictions to downstream tables or applications, and monitor the entire process to avoid performance degradation and instability.

**Data scientists** typically do not have write or compute access in the production environment. However, it is important that they have visibility to test results, logs, model artifacts, production pipeline status, and monitoring tables. This visibility allows them to identify and diagnose problems in production and to compare the performance of new models to models currently in production.



1. **Train model:**

This pipeline can be triggered by code changes or by automated retraining jobs.

* **Training and tuning**. During the training process, logs are recorded to the production environment **MLflow Tracking server**. These logs include model metrics, parameters, tags, and the model itself. If you use feature tables, the model is logged to **MLflow** using the **Databricks Feature Store client**, which packages the model with feature lookup information that is used at inference time.

During development, **data scientists** may test many algorithms and hyperparameters. In the production training code, it’s common to consider only the top-performing options. Limiting tuning in this way saves time and can reduce the variance from tuning in automated retraining.

If data scientists have read-only access to the production catalog, they may be able to determine the optimal set of hyperparameters for a model. In this case, the model training pipeline deployed in production can be executed using the selected set of hyperparameters, typically included in the pipeline as a configuration file.

* **Evaluation.** Model quality is evaluated by testing on held-out production data. The results of these tests are logged to the **MLflow tracking server**. This step uses the evaluation metrics specified by **data scientists** in the development stage. These metrics may include custom code.
* **Register model.** When model training is complete, the model artifact is saved as a registered model version at the specified model path in the production catalog in **Unity Catalog**. The model training task yields a model URI that the model validation task can use. You can use task values to pass this URI to the model.

1. **Validate model:**

* This pipeline uses the model URI from Step 1 and loads the model from **Unity Catalog**. It then executes a series of validation checks. These checks depend on your organization and use case, and can include things like basic format and metadata validations, performance evaluations on selected data slices, and compliance with organizational requirements such as compliance checks for tags or documentation.
* If the model successfully passes all validation checks, you can assign the “**Challenger**” alias to the model version in **Unity Catalog**. If the model does not pass all validation checks, the process exits and users can be automatically notified. You can use tags to add key-value attributes depending on the outcome of these validation checks. For example, you could create a tag “model\_validation\_status” and set the value to “PENDING” as the tests execute, and then update it to “PASSED” or “FAILED” when the pipeline is complete.
* Because the model is registered to **Unity Catalog**, **data scientists** working in the development environment can load this model version from the production catalog to investigate if the model fails validation. Regardless of the outcome, results are recorded to the registered model in the production catalog using annotations to the model version.

1. **Deploy model:**

This section assumes that you have assigned the newly validated model the “**Challenger**” alias, and that the existing production model has been assigned the “**Champion**” alias. The first step before deploying the new model is to confirm that it performs at least as well as the current production model.

* **Compare “CHALLENGER” to “CHAMPION” model.** You can perform this comparison offline or online. An offline comparison evaluates both models against a held-out data set and tracks results using the **MLflow Tracking server**. For real-time model serving, you might want to perform longer running online comparisons, such as **A/B tests** or a gradual rollout of the new model. If the “**Challenger**” model version performs better in the comparison, it replaces the current “**Champion**” alias.

**Mosaic AI Model Serving** and **Databricks Lakehouse Monitoring** allow you to automatically collect and monitor inference tables that contain request and response data for an endpoint.

If there is no existing “**Champion**” model, you might compare the “**Challenger**” model to a business heuristic or other threshold as a baseline.

The process described here is fully automated. If manual approval steps are required, you can set those up using workflow notifications or CI/CD callbacks from the model deployment pipeline.

* **Deploy model.** Batch or streaming inference pipelines can be set up to use the model with the “**Champion**” alias. For real-time use cases, you must set up the infrastructure to deploy the model as a **REST API endpoint**. You can create and manage this endpoint using **Mosaic AI Model Serving**. If an endpoint is already in use for the current model, you can update the endpoint with the new model. **Mosaic AI Model Serving** executes a zero-downtime update by keeping the existing configuration running until the new one is ready.

1. **Model Serving:**

* When configuring a **Model Serving endpoint**, you specify the name of the model in **Unity Catalog** and the version to serve. If the model version was trained using features from tables in **Unity Catalog**, the model stores the dependencies for the features and functions. **Model Serving** automatically uses this dependency graph to look up features from appropriate online stores at inference time. This approach can also be used to apply functions for data preprocessing or to compute on-demand features during model scoring.
* You can create a single endpoint with multiple models and specify the endpoint traffic split between those models, allowing you to conduct online “**Champion**” versus “**Challenger**” comparisons.

1. **Inference: batch or streaming**

* The inference pipeline reads the latest data from the production catalog, executes functions to compute on-demand features, loads the “**Champion**” model, scores the data, and returns predictions. Batch or streaming inference is generally the most cost-effective option for higher throughput, higher latency use cases. For scenarios where low-latency predictions are required, but predictions can be computed offline, these batch predictions can be published to an online key-value store such as **DynamoDB** or **Cosmos DB**.
* The registered model in **Unity Catalog** is referenced by its alias. The inference pipeline is configured to load and apply the “**Champion**” model version. If the “**Champion**” version is updated to a new model version, the inference pipeline automatically uses the new version for its next execution. In this way the model deployment step is decoupled from inference pipelines.
* Batch jobs typically publish predictions to tables in the production catalog, to flat files, or over a JDBC connection. Streaming jobs typically publish predictions either to **Unity Catalog tables** or to message queues like **Apache Kafka**.

1. **Lakehouse Monitoring:**

Lakehouse Monitoring monitors statistical properties, such as data drift and model performance, of input data and model predictions. You can create alerts based on these metrics or publish them in dashboards.

* **Data ingestion**. This pipeline reads in logs from batch, streaming, or online inference.
* **Check accuracy and data drift**. The pipeline computes metrics about the input data, the model’s predictions, and the infrastructure performance. **Data scientists** specify data and model metrics during development, and **ML engineers** specify infrastructure metrics. You can also define custom metrics with Lakehouse Monitoring.
* **Publish metrics and set up alerts**. The pipeline writes to tables in the production catalog for analysis and reporting. You should configure these tables to be readable from the development environment so **data scientists** have access for analysis. You can use **Databricks SQL** to create monitoring dashboards to track model performance, and set up the monitoring job or the dashboard tool to issue a notification when a metric exceeds a specified threshold.
* **Trigger model retraining**. When monitoring metrics indicate performance issues or changes in the input data, the **data scientist** may need to develop a new model version. You can set up SQL alerts to notify data scientists when this happens.

1. **Retraining:**

This architecture supports automatic retraining using the same model training pipeline above. Databricks recommends beginning with **scheduled**, **periodic retraining** and moving to **triggered retraining when needed**.

* **Scheduled.** If new data is available on a regular basis, you can create a scheduled job to run the model training code on the latest available data.
* **Triggered.** If the monitoring pipeline can identify model performance issues and send alerts, it can also trigger retraining. For example, if the distribution of incoming data changes significantly or if the model performance degrades, automatic retraining and redeployment can boost model performance with minimal human intervention. This can be achieved through a **SQL alert** to check whether a metric is anomalous (for example, check drift or model quality against a threshold). The alert can be configured to use a **webhook** destination, which can subsequently trigger the training workflow.

If the retraining pipeline or other pipelines exhibit performance issues, the **data scientist** may need to return to the development environment for additional experimentation to address the issues.

# Material:

Material: <https://docs.databricks.com/en/lakehouse-monitoring/index.html>

MLOps Tutorial: <https://www.databricks.com/resources/demos/tutorials/data-science-and-ai/mlops-end-to-end-pipeline?itm_data=demo_center>

Training: <https://customer-academy.databricks.com/learn/catalog?ctldoc-catalog-0=p-0>

# MLOps Random Forest Forecasting:

**Using Random Forest for Sales Forecasting**

Databricks: <https://adb-3978578346428464.4.azuredatabricks.net/browse?o=3978578346428464>

Dataset: <https://www.kaggle.com/datasets/jr2ngb/superstore-data>

Reference: <https://medium.com/@tebugging/databricks-end-to-end-machine-learning-create-an-ingest-to-serving-mlops-pipeline-2f739f90e65f>

No Unity Catalog: <https://docs.databricks.com/en/_extras/notebooks/source/mlflow/mlflow-end-to-end-example.html>

Step 1: Set Up Databricks Environment

1. Create a new workspace:

* Set up workspace
* Create a new cluster or use default cluster

1. Upload the Dataset:

* In Databricks, go to the "Catalog" tab, and then click "Add Data" to upload your dataset.
* Start by creating a new Databricks notebook.
* Ensure your data is accessible through the Databricks Catalog.

Step 2: Load and Explore the Data

1. Load the Data from Catalog into a DataFrame:

* Load Dataset
* Number of rows and columns
* Data types for each column
* Summary statistics for dataset

1. Perform data cleaning and data preprocessing:

* Checking duplicates
* Remove unnecessary columns / Feature Selection
* Checking null values (Imputation if necessary)
* Rename column names
  + - Check the format of specific data types and Transform data types
    - Create new features for feature engineering

Step 3: EDA Visualization

1. Autocorrelation
   * + Helps to identify patterns or trends in the data by checking how related the current values are to previous values at different time lags
     + Horizontal dashed lines represent the confidence intervals (often 95%). If the autocorrelation value is within these lines, it is not statistically significant, meaning there's no strong evidence of correlation at that lag.
     + If you choose lags that have significant positive autocorrelation, it means you are capturing important temporal dependencies, which can improve the model's performance.
2. Line Chart
   * + The total sales have been increasing every year, with a wide range. In 2011, total sales ranged from 100,000 to 300,000, but by 2014, they exceeded 500,000. This large range might be why the Random Forest model cannot forecast precisely.
     + The average sales from 2011 to 2014 are quite consistent, ranging between 210 and 280. However, in some months, the average sales fluctuate drastically.
3. Lag plot
   * + If the lag plot gives a linear plot, then it means the autocorrelation is present in the data, whether there is positive autocorrelation or negative that depends upon the slope of the line of the dataset. If more data is concentrated on the diagonal in lag plot, it means there is a strong autocorrelation.
     + The lag plot is also useful for checking whether the given dataset is random or not. If there is randomness in the data then it will be reflected in the lag plot, if there is no pattern in the lag plot.
     + The points are somewhat scattered, indicating that while there is a trend, it is not a perfect linear relationship. There is some degree of variability or noise in the data.

Some models, especially those with built-in complexity like **Random Forests**, Gradient Boosting, or advanced time series models like Prophet, can handle data with weak or no clear autocorrelation by learning from the available data in a non-linear fashion.

Step 4: Feature Engineering

* + - Create lag features for time series data, number of lags getting from the autocorrelation
    - Remove null values that created from the lag features

Step 5: Modelling

1. Fine tune model / Normalization
   * + Grid Search to fine tune the model
2. Define X and y
3. Split data
4. Train model
   * + Train on Random Forest model
     + Set seed number and n\_estimator
5. Test model
6. Evaluate model
   * + RMSE (Root Mean Squared Error): 18.406539339764727, The average difference between your model's predictions and the actual values is around 18.406539339764727.
     + Mean Absolute Error (MAE): 15.23723403125409, MAE represents the average absolute difference between predicted values and actual values. The MAE is lower than the RMSE, which indicates that the model might have some larger errors that the RMSE is capturing
     + R-squared: -1.146062560508613, R-squared measures how well your model explains the variability of the target variable. A negative R-squared is a strong indicator that your model is not capturing the underlying patterns in the data effectively. R-squared is generally more appropriate when the focus is on explaining the variance in the data, particularly when the data is linear.
     + MAPE: 0.06490659892435668, MAPE can be used to find the percentage error between any predicted and real value. MAPE can be applied to any forecasting system, including generic regression problems and time series data. MAPE is a quick and easy accuracy test to perform and takes mere seconds for most datasets. A MAPE under 10% shows high accuracy, 10-20% shows good forecast results, 20-50% shows reasonable forecast results, and greater than 50% shows incorrect results.
     + Cross-validation MAE scores: [158.70134892 460.54660561 384.15971386 360.48321615 500.90874148], These scores show the MAE for each fold in your cross-validation. Cross-validation helps assess how well your model generalizes to unseen data. The scores vary significantly (from ~160 to ~501), indicating that the model's performance is inconsistent across different subsets of the data.
     + Mean cross-validation MAE: 372.95992520480445, The average MAE across all cross-validation folds is 372.95992520480445. If the cross-validation MAE is much higher than the regular MAE, it suggests that the model may be overfitting the training data. The model performs well on the specific test set but poorly on unseen data during cross-validation, implying it might not generalize well.