2ND EDITION

eBook

The Big Book of MLOps

NOW INCLUDING A SECTION ON LLMOPS

MODELOPS + DATAOPS + DEVOPS

databricks

JOSEPH BRADLEY | RAFI KURLANSIK | MATT THOMSON | NIALL TURBITT

Contents

databricks

Contents

databricks

Contents

CHAPTER 1

Introduction

Machine learning operations (MLOps) is a rapidly evolving field where building and maintaining robust, flexible and efficient workflows is critical. At Databricks, we view MLOps as the set of processes and automation for managing data, code and models to improve performance stability and long-term efficiency in ML systems. Distilling this into a single equation:

> MLOps = DataOps + DevOps + ModelOps

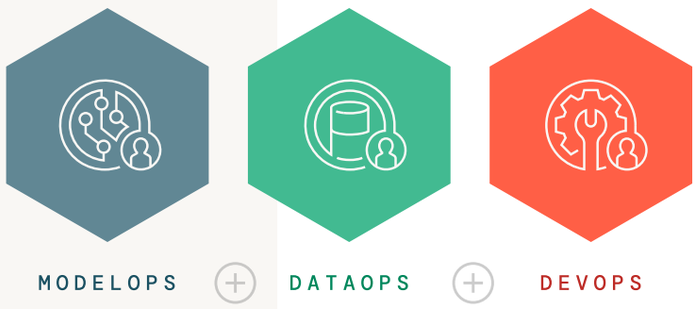
Through this lens, we strive to continuously innovate and advance our product offerings to simplify the ability to build AI-powered solutions on the Lakehouse. We believe there is no greater accelerant to delivering ML to production than building on a unified, data-centric AI platform. On Databricks, both data and models can be managed and governed in a single governance solution in the form of Unity Catalog. The previously complex infrastructure required to serve real-time models can now be replaced and easily scaled with Databricks Model Serving. Long-term efficiency and performance stability of ML in production can be achieved using Databricks Lakehouse Monitoring. These components collectively form the data pipelines of an ML solution, all of which can be orchestrated using Databricks Workflows.

Perhaps the most significant recent change in the machine learning landscape has been the rapid advancement of generative AI. Generative models such as large language models (LLMs) and image generation models have revolutionized the field, unlocking previously unattainable levels of natural language and image generation. However, their arrival also introduces a new set of challenges and decisions to be made in the context of MLOps.

With all these developments in mind, we're excited to present this updated version of the Big Book of MLOps. This guide incorporates new Databricks features such as Models in Unity Catalog, Model Serving, and Lakehouse Monitoring into our MLOps architecture recommendations. We start by outlining the themes that still remain relevant from the previous version of the Big Book of MLOps. Following this, we unpack the new features introduced in this version, their impact on the previous reference architecture, and best practices when incorporating these into your MLOps workflows. Next, we present our updated MLOps reference architecture, along with the details of its processes. Finally, we provide guidance for deploying generative AI applications to production on Databricks, focusing on productionizing LLMs.

CHAPTER 2

Big Book of MLOps V1 Recap



We begin with a brief recap of the core points discussed in the previous version of the Big Book of MLOps. While the recommended reference architecture has evolved due to new features and product updates, the core themes discussed, such as the importance of MLOps, guiding principles and the fundamentals of MLOps on Databricks, remain pertinent. In this section we focus on summarizing those elements that remain unchanged. For a more in-depth discussion of any of these points, we refer the reader to last year's Big Book of MLOps.

Why should I care about MLOps?

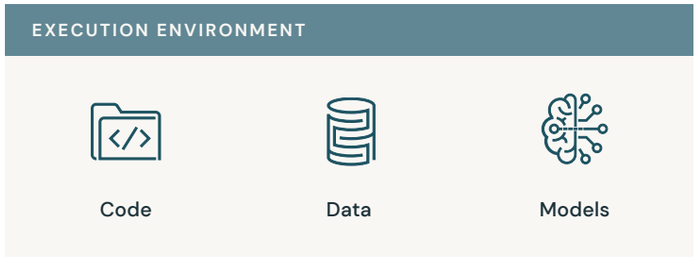
We continue to stress the importance of defining an effective MLOps strategy. Databricks customers like CareSource, which has since implemented our recommended MLOps architecture, have witnessed firsthand the value this can bring. Through streamlining the process of delivering models to production, time to business value is accelerated. This efficiency has the knock-on effect of giving data science teams the freedom and confidence to transition to subsequent projects without the need for continuous manual oversight of models in production.

Guiding principles

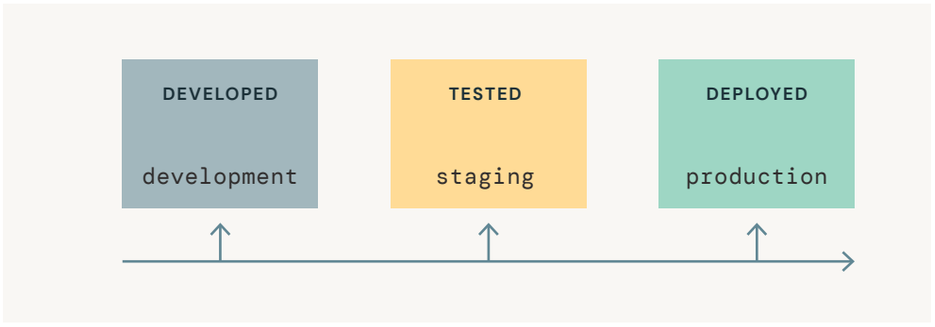
One guiding principle that continues to lie at the heart of the Lakehouse AI vision is taking a data-centric approach to machine learning. With the increasing prevalence of generative AI, this perspective remains just as important. The core constituents of any ML project can be viewed simply as data pipelines: feature engineering, training, model deployment, inference and monitoring pipelines are all data pipelines. As such, operationalizing an ML solution requires joining data from predictions, monitoring and feature tables with other relevant data. Fundamentally, the simplest way to achieve this is to develop AI-powered solutions on the same platform used to manage production data.

Note: Throughout this paper we operate under the assumption of three distinct execution environments — development, staging and production — in the form of three separate Databricks workspaces. There can be variations of these three stages, such as alternative naming conventions or splitting staging into separate "test" and "QA" substages. Although not recommended, it is also possible to create three distinct environments within a single Databricks workspace through the use of access controls and Git branches. Regardless of how environment separation is achieved, the core principles of the workflow and recommendations presented are generally applicable.

Semantics of development, staging and production



An ML solution comprises data, code and models. These assets need to be developed, tested (staging) and deployed (production). For each of these stages, we also need to operate within an execution environment. As such, each of data, code, models and execution environments are notionally divided into development, staging and production.

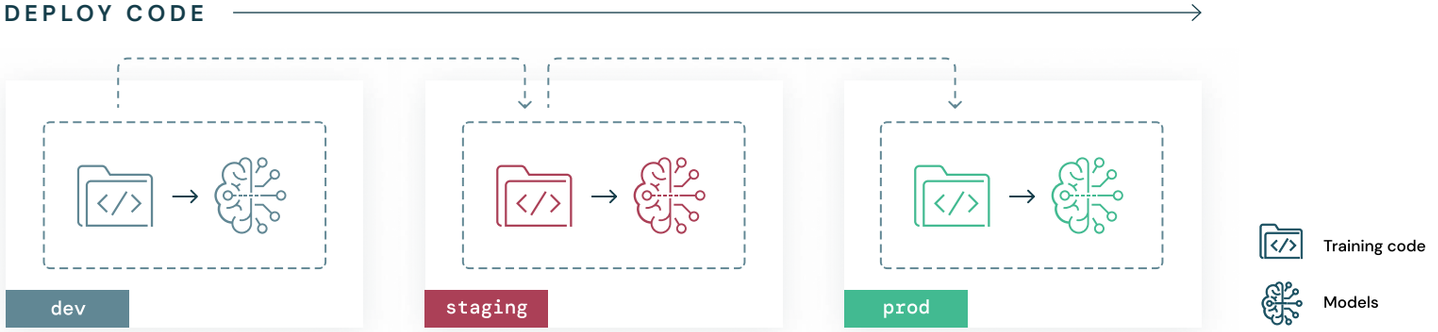


Each of these stages has distinct access controls and quality guarantees, ranging from the open and exploratory development stage through to the locked-down and quality-assured production stage.

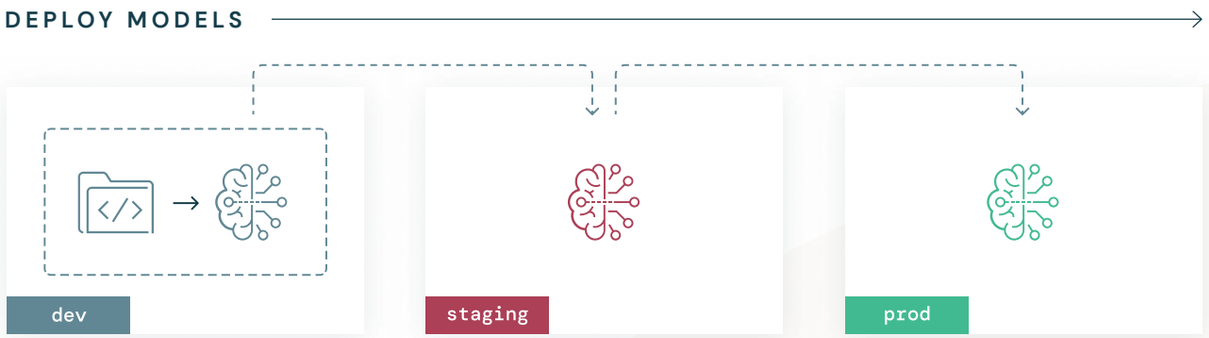
ML deployment patterns

Code and models often progress asynchronously through these stages. Thus, it becomes crucial to leverage a solution that allows for the management of model artifacts independently of code, making it possible to update a production model without necessarily making a code change. Data, much like code and models, can be labeled as development, staging or production, indicating not only its origin but also its quality and reliability.

Given the independent lifecycles of code and models, there are two opposing strategies to moving code and ML models from development, through staging and subsequently to production:



- Code for an ML project is developed in the development environment, and this code is then moved to the staging environment, where it is tested. Following successful testing, the project code is deployed to the production environment, where it is executed.
- Model training code is tested in the staging environment using a subset of data, and the model training pipeline is executed in the production environment
- The model deployment process of validating a model and additionally conducting comparisons versus any existing production model all run within the production environment



- Model training is executed in the development environment. The produced model artifact is then moved to the staging environment for model validation checks, prior to deployment of the model to the production environment.
- This approach requires a separate path for deploying ancillary code such as inference and monitoring code. Subsequently, any pipelines that need to run in the production environment to support the operationalization of the model will necessarily need to go through a separate "deploy code" lifecycle — the code for these components being tested in staging and then deployed to production.
- This pattern is typically used when deploying a one-off model, or when model training is expensive and read-access to production data from the development environment is possible

As in our prior paper, we **recommend a deploy code approach for the majority of use cases**, and the reference architecture presented in this update continues to follow this recommendation.

CHAPTER 3

What's New?

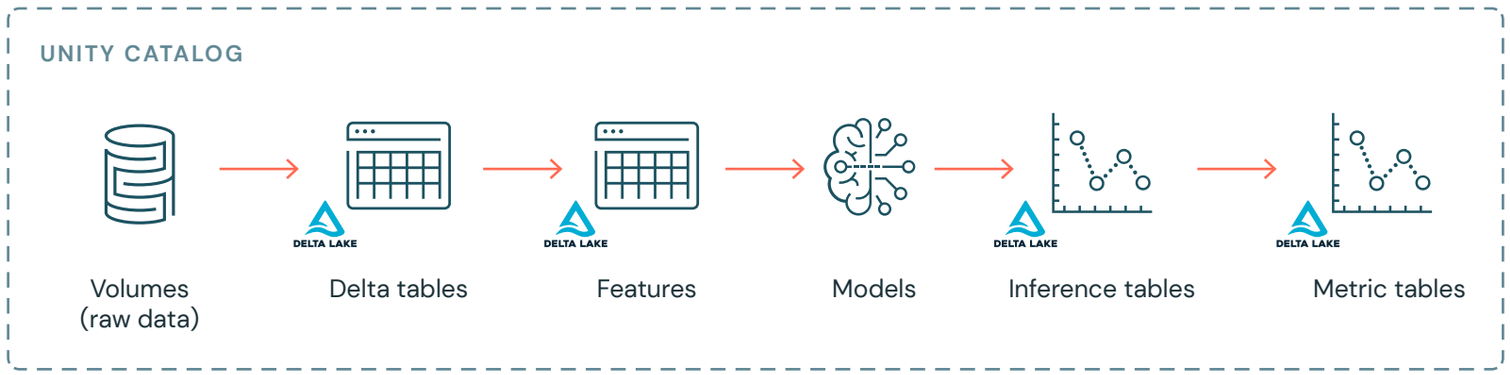
In this section we outline the key features and product updates introduced into our updated MLOps reference architecture. For each of these, we highlight the benefits they bring and how they impact our end-to-end MLOps workflow.

Unity Catalog

The Lakehouse forms the foundation of a data-centric AI platform. Key to this is the ability to manage both data and AI assets from a unified governance solution on the Lakehouse. Databricks Unity Catalog enables this by providing centralized access control, auditing, lineage, and data discovery capabilities across Databricks workspaces.

These benefits are now extended to MLflow models with the introduction of Models in Unity Catalog. By providing a hosted version of the MLflow Model Registry in Unity Catalog, the full lifecycle of an ML model can be managed while leveraging Unity Catalog's capability to share assets across Databricks workspaces and trace lineage across both data and models.

In addition to managing ML models, feature tables are also a part of Unity Catalog. With Feature Engineering in Unity Catalog, any Delta table in Unity Catalog that has been assigned a primary key (and additionally a timestamp key) can be used as a source of features to train and serve models. Furthermore, feature tables can now also be shared across different workspaces, and lineage recorded between other assets in the Lakehouse.



Assets of an ML workflow, all managed via Unity Catalog

BENEFITS AND ARCHITECTURE IMPLICATIONS

Unified governance

- Unity Catalog unlocks the ability to apply the same governance and security policies to both data and models. Consolidating these assets into a single, unified solution facilitates efficient and secure management, and more generally simplifies the overall MLOps solution.
- From an architecture design standpoint, this means the flexibility to govern both data and AI assets under the same namespace, enabling management at the environment, project or team level.

Read access to production assets

- With Unity Catalog, data and models are governed at the account level, promoting easy sharing of assets across Databricks workspaces. Data scientists in the development environment can be granted read-only access to data and AI assets from the production environment. The specifics of cross-workspace permissions can be adjusted for each project or organization.
- The implications of this are multifold: Data scientists can train models using production data, detect and debug model quality degradation by examining production monitoring tables, deep dive on model predictions using production inference tables, and easily compare in-development models with live production models. This not only accelerates the model development process but also improves the robustness and quality of the developed models.

Models in Unity Catalog

- It is now recommended to register Models in Unity Catalog in place of the classic Workspace Model Registry. Models will be registered under a three-level name in the form `<catalog>.<schema>.<model>` (see Organizing data and AI assets for a more detailed walkthrough of this).
- This three-level name provides the ability to inherently express the environment a model was produced in and apply associated governance permissions at each level of the catalog/schema/registered model hierarchy. For example, permissions can be configured such that all users have read-only access to models in the prod catalog, only ML team service principals can create new models in the use case–specific prod.fraud_detection schema, and all members of the fraud detection team can execute model versions in the `prod.fraud_detection.fraud_clf` registered model.

- The model alias — a mutable, named reference to a version of a registered model — can be used to mark a model for deployment. Inference workloads can be defined to target a specific alias. For example, you could assign the "Champion" alias of the "fraud_clf" registered model to the model version that should serve production traffic. The inference workload of the fraud detection solution would then target that alias, using the "Champion" model to make predictions.

Lineage

- With Unity Catalog, a robust link between data and AI assets can natively be recorded. Lineage can be traced from a model version in Unity Catalog back to the data used for training. Additionally, downstream lineage records consumers of assets in Unity Catalog. For example, consumers of a registered model could include Model Serving endpoints or Workflows, facilitating impact analysis if a regression is detected in the model.
- This lineage also extends from a model version to the underlying MLflow run in MLflow Tracking. Parameters, metrics, artifacts and Git information can all be tracked to this MLflow run. Additionally, metadata detailing data sources, notebooks, jobs and endpoints associated with the assets will be recorded automatically for both tables and models stored in Unity Catalog. Designing an MLOps implementation to leverage MLflow during model training ensures that this full lineage can be captured from the beginning of a model's lifecycle.

Discoverability

- Through centralizing data and AI assets in a single solution, Unity Catalog enhances the discoverability of these assets, making it simpler and quicker to locate and utilize the appropriate resources for a particular component of the MLOps solution. For example, teammates with access to view each other's models can see which data sources are being used, and use that information to address their own use cases.

Model Serving

Databricks Model Serving provides a production-ready, serverless solution to simplify real-time ML model deployment. With Model Serving, it is possible to efficiently deploy models as an API so that you can integrate model predictions with applications or websites. Given the complexity that is often involved with deploying a real-time ML model, Model Serving reduces operational costs, streamlines the ML lifecycle, and makes it easier for data science teams to focus on the core task of integrating production-grade real-time ML into their solutions.

BENEFITS AND ARCHITECTURE IMPLICATIONS

Lakehouse native

- Model Serving natively integrates with other components of the Lakehouse platform, such as Unity Catalog and MLflow. When a registered MLflow model is served, request-response payloads from Model Serving endpoints can be automatically logged to inference tables in Unity Catalog. In addition to the lineage captured from an inference table back to a registered model, we also gain the ability to implement model monitoring (see Lakehouse Monitoring). From an operational perspective, these production inference tables can be made available to data scientists in the development environment.
- Given its close integration with MLflow, Model Serving can automatically build a container from a logged MLflow model and deploy the model as a REST endpoint. This abstracts away what would ordinarily be a notably more complex process for the user.
- If models are trained using ML features from Unity Catalog, Model Serving uses lineage in Unity Catalog to automatically serve features for batch and online serving

Simplified deployment

- Data scientists or ML engineers can easily create a Model Serving endpoint from a model version without requiring extensive infrastructure knowledge or experience. Thus, creating or updating a Model Serving endpoint becomes a trivial additional step in the model deployment pipeline.

High availability and scalability

- Built for production use, Model Serving supports very low latency (p50 overhead latency of less than 10ms) and high query volumes (QPS of greater than 25k), and can automatically scale up and down based on demand. This ensures optimal performance and cost-efficiency.
- Model deployment configurations should be robustly tested prior to exposing a real-time model endpoint to production traffic to meet predefined service level agreements (SLAs). As such, we must factor such tests into our MLOps reference architecture. We unpack the different types of testing that can be conducted below in Testing Model Serving. We also illustrate how this testing is incorporated into an MLOps workflow in the Reference Architecture section.

Online evaluation

- Serving real-time models often involves online evaluation, in addition to standard offline model evaluation approaches typical of batch or streaming inference pipelines. Your choice of online evaluation strategies such as A/B testing or canary deployments can affect the design and implementation of pipelines being deployed to production. Model Serving facilitates the implementation of such online evaluation approaches through the ability to serve multiple models to a Model Serving endpoint.
- The design decisions taken around online evaluation will depend on use case requirements. We explore these in more detail in the Model deployment section below.

Secure and cost effective

- Models are deployed in a secure network boundary. Model Serving leverages serverless compute that will terminate when the model is deleted, or scaled down to zero.

Lakehouse Monitoring

Databricks Lakehouse Monitoring is a data-centric monitoring solution to ensure that both data and AI assets are of high quality, accurate and reliable. Built on top of Unity Catalog, it provides the unique ability to implement both data and model monitoring while maintaining lineage between the data and AI assets of an MLOps solution. This unified and centralized approach to monitoring greatly simplifies the process of diagnosing errors, detecting quality drift and performing root cause analysis.

BENEFITS AND ARCHITECTURE IMPLICATIONS

Lakehouse native

- Lakehouse Monitoring integrates with Unity Catalog to automatically write monitoring tables to the Lakehouse — computed metrics are stored in Delta tables called metric tables. All the advantages of Unity Catalog as outlined previously thus apply. A key benefit is the streamlined process of joining monitoring tables with others in the Lakehouse, providing an efficient way to cross-analyze data. Users can perform these analyses using SQL, and present results in both dashboards and notebooks.
- The core aim of Lakehouse Monitoring is to expedite the ability to detect and diagnose deviations in data or model quality. Thus, configuring appropriate permissions on production monitoring tables and dashboards such that they are accessible to data scientists should be factored into the production deployment process.
- To keep monitoring tables up to date, Lakehouse Monitoring APIs allow you to schedule a refresh of the metric tables on a regular basis

Monitoring with Model Serving

- Lakehouse Monitoring has tight integration with Model Serving, in particular allowing the ability to monitor inference tables produced by a Model Serving endpoint. With this integration, a pipeline can be created to process logged requests and responses from a Model Serving inference table, join with other relevant tables in the Lakehouse (e.g., labels or business metrics) and run Lakehouse Monitoring on the resulting table to produce data and model quality metrics.

Simplification

- Lakehouse Monitoring provides a single monitoring tool and experience, regardless of whether monitoring data or AI. At present this is the only tool that supports a common API for monitoring both models and data.

Customization

- Users can introduce custom metrics based on specific business needs, further enhancing monitoring capabilities. Custom metrics can include aggregates or drift metrics that adhere to specific business logic.
- Additionally, it is possible to examine model performance on given data slices. This is often crucial to catch errors in subpopulations of interest, or when evaluating bias/fairness.

Dashboards and alerts

- Lakehouse Monitoring automatically generates a Databricks SQL dashboard for visualizing computed monitoring metrics. Additionally, this generated dashboard has user-editable parameters for both the whole dashboard and individual charts, allowing users to customize the date range, slicing logic, model versions, etc. Users can also add their own charts to the dashboard that join the monitoring metrics with external business data.
- Alerts can be defined against metrics in the generated metrics tables. A natural evolution to this is to set up alerts when quality or performance indicators deviate from expectations. This can be achieved using Databricks SQL alerts operating on a defined Databricks SQL query. The definition of this alerting criteria, and frequency of evaluation, will subsequently become part of the code deployed to the production environment.

CHAPTER 4

Design Decisions

Before presenting our updated reference architecture, let's highlight a number of design considerations you should take into account when first architecting an MLOps solution on Databricks.

Unity Catalog

ORGANIZING DATA AND AI ASSETS

One architectural decision that extends beyond a single MLOps solution is defining how both data and AI assets are organized within Unity Catalog. Below are a few factors motivating why thoughtful organization of AI assets within Unity Catalog is essential.

Efficiency and accessibility

- With a growing number of AI assets, organizations can face a situation where data and models are spread across different platforms and storage systems. This dispersion can make it difficult to find, manage and govern AI assets.
- A well-structured organization of data and AI assets within Unity Catalog ensures that data scientists, ML engineers and other stakeholders can easily locate and access the resources they need. This reduces the time spent searching for specific models or tables, and allows more time for actual use case development.

Scalability

- As an organization's data needs grow, so too does the number of AI assets. A well-architected Unity Catalog structure helps manage this growth, ensuring that the number of use cases can scale smoothly without becoming unmanageable.
- Having a well-defined structure to store AI assets further facilitates the addition of new data and ML models without disrupting existing use cases in production
- Further, a well-defined MLOps workflow enforces a standardized approach to deploying AI assets to production using Unity Catalog. This facilitates scalability to many use cases within an organization.

Governance

- By categorizing and structuring data and AI assets in a well-defined manner, organizations can implement effective access controls using Unity Catalog, ensuring that only authorized individuals can access certain data or models. In many cases this is required for compliance with data protection regulations, and for maintaining data privacy and security.
- For the purposes of auditability, Unity Catalog captures a log of actions performed against the metastore, and these logs are delivered as part of Databricks audit logs.

Collaboration

- In many organizations, multiple teams or individuals will be working with the same AI assets — sharing features and models across different use cases
- Consolidating both data and AI assets into a centralized location, and clearly delineating where specific assets are located and how they should be used, facilitates collaboration

CONCEPTS

Unity Catalog has a number of core concepts that we should define before unpacking the considerations when organizing AI assets along with data in Unity Catalog.

Catalog

Akin to a database. It serves as a container for all your data and AI assets, such as tables and models.
A catalog can be thought of as a namespace for these data and AI assets. Later, in our proposed architecture, we will illustrate separate "dev," "staging" and "prod" catalogs. These catalogs will contain data and AI models corresponding to the environment in which they were produced.

Schema

A logical construct within a catalog that groups related tables, views and models together. Put simply, it's a way to organize related assets within a catalog.

Data tables

Data tables are the leaf level assets in Unity Catalog and can be referenced using a three-level name provided in the form `<catalog>.<schema>.<table>` . Any Delta table that has a primary key (and optionally a time series key) can be used as a source of ML features which can be joined with training samples to train an ML model.

Volume

Represents a logical volume of storage in a cloud object storage location. Volumes provide capabilities for accessing, storing, governing and organizing files. While tables provide governance over tabular data sets, volumes add governance over non-tabular data sets. You can use volumes to store and access files in any format, including structured, semi-structured and unstructured data. A volume will reside within a schema, under `<catalog>.<schema>.<volume-name>`.

Functions

SQL and Python UDFs are a part of Unity Catalog and can be accessed using a three-level name provided in the form `<catalog>.<schema>.<function>`. Functions enable the ability to use Python UDFs to compute on-demand features as inputs for machine learning, and can be used in model training, batch inference and real-time inference. Functions are appropriate in cases where fresh features (e.g., computing a user's distance to a restaurant) need to be computed at inference time, or where features from data sources cannot be pre-materialized to a data lake for regulatory or security reasons (e.g., a user's credit score).

Registered model

An MLflow model that has been registered to Unity Catalog. The registered model has a unique name, versions, model lineage and other metadata. When registering a model to Unity Catalog, a three-level name is provided in the form `<catalog>.<schema>.<model>`.

Model version

A version of a registered model. When a new model is added to the Model Registry, it is added as Version 1. Each model registered to the same model name increments the version number. A specific model version can be loaded using the model URI `models:/<catalog>.<schema>.<model>/<model_version>`.

Model alias

A mutable, named reference to a particular version of a registered model. Typical uses of aliases are to specify which model version should be deployed in a model deployment pipeline, or to write inference workloads that target a specific alias. Aliases are flexible, and as such can be tailored to suit your use case or organization requirements. For example, you could assign the model version that should serve the majority of production traffic the "Champion" alias. Inference workloads could then target that alias using the model URI `models:/<catalog>.<schema>.<model>@Champion`.

CONSIDERATIONS

When designing how to organize your data and AI assets in Unity Catalog, it's worth considering how to best leverage the three-level namespace of catalog, schema and entity name. For example:

1. Do you equate each schema to a different business unit or team, under which tables and models specific to that business unit or team reside?
2. Do you have schemas within a catalog mapped to a medallion architecture of Bronze, Silver and Gold tables? If so, where do feature tables and ML models then reside?

While there is no one-size-fits-all approach to managing AI assets with data in Unity Catalog, there are a number of guiding principles that can be applied when contending with these decisions:

- **Team size**
  The size of a team can influence the level of detail in the organizational structure of data and AI assets in Unity Catalog. Larger teams might require more granular categorizations (e.g., specifying a schema per business unit or team), while smaller teams might function well with a simpler structure (e.g., all teams operating across shared schemas — Bronze, Silver, Gold — within each catalog).

- **Complexity of projects**
  More complex projects, involving the registration of many different models, may necessitate dedicated schemas per project. It should be noted, however, that this requirement can be handled in some cases through the creation of a custom MLflow PyFunc model. This approach can be used to create a wrapper around many models using a single PyFunc model, which can then be registered in the same fashion as any other individual model.

- **Access levels and permissions**
  Consideration should be given to the different access levels and permissions that will be required. Some assets might need to be accessible to everyone on a given team, while others should be restricted to specific individuals or roles. With Unity Catalog, both data and AI assets can be grouped and governed at the environment, project or team level.

- **Models, functions and features in Unity Catalog**
  With Models in Unity Catalog, AI artifacts can be governed in the same way as data in the Lakehouse. This unified governance enables the sharing of models across Databricks workspaces, and teams within an organization. As such, where models, functions and features reside within a given catalog is often determined by organizational requirements around sharing of these assets across teams and use cases. From a catalog perspective, Models in Unity Catalog should be treated no differently than data in Unity Catalog — models are registered to a catalog indicating the environment in which they were produced. Aliases can then provide the ability to indicate which models are deployed in which contexts (e.g., a "Champion" model serves the majority of production traffic, while the "Challenger" model serves a small fraction for testing).

- **Discoverability**
  A standardized approach across teams in how assets are organized within Unity Catalog is important for setting up ACLs and discoverability. Notably, there are also methods like tags to aid discoverability.

RECOMMENDED ORGANIZATION

While there is no universal blueprint for organizing data and AI assets in Unity Catalog, the structure we propose below serves as a robust starting point. It's designed with flexibility in mind, providing a clear demarcation between assets created in different environments while also enabling simplified discoverability, sharing, and governance of data and AI assets across teams. Notably, we replicate the structure across dev, staging and prod catalogs. This enables users to develop and test against catalog structures — dev and staging, respectively — mirroring that of the production environment. For this example, we show all ML assets for a specific "fraud detection" use case organized under a single "fraud_detection" schema.

```
dev
bronze
    > Tables:
silver
    > Tables:
gold
    > Tables:
      user_features
      product_features
fraud_detection
    > Tables:
      offline_location_features
      fraud_clf_inference
      fraud_clf_metrics
    > Volumes:
    > Models:
      fraud_clf
    > Functions:
      compute_distance
```

```
staging
bronze
    > Tables:
silver
    > Tables:
gold
    > Tables:
      user_features
      product_features
fraud_detection
    > Tables:
      fraud_clf_inference
      fraud_clf_metrics
    > Volumes:
    > Models:
      fraud_clf
    > Functions:
      compute_distance
```

```
prod
bronze
    > Tables:
silver
    > Tables:
gold
    > Tables:
      user_features
      product_features
fraud_detection
    > Tables:
      fraud_clf_inference
      fraud_clf_metrics
    > Volumes:
    > Models:
      fraud_clf
    > Functions:
      compute_distance
```

Catalog level

At the catalog level, assets are segregated based on the environment from which they were produced. As a result, teams have an inherent understanding of the maturity and readiness of each asset.

`dev catalog`

- Assets under active development and testing. We illustrate additional tables and models within this catalog, indicative of new assets in development.
- Relatively open access, enabling users to easily read and write assets during the development process

`staging catalog`

- Used as an area for writing assets produced during the testing of code in the staging environment prior to deployment to the production environment
- Additionally, this catalog could also be used to store more permanent preproduction assets for the purposes of mirroring the production environment during integration testing
- Data and models stored in the staging catalog may be temporary, or cleaned up on some periodic basis
- Limited write access aside from that of administrators and service principals
- Read access should be enabled for users who may need to debug integration tests

`prod catalog`

- Assets that have been produced using production-deployed code
- We can inherently assert that these assets have been produced by code that has been tested prior to deployment to the production environment
- Access to write to the prod catalog is typically limited to a small number of administrators or service principals
- Access to read from the prod catalog can be granted to users in non-production Databricks workspaces

Schema level

Within each catalog, assets are further categorized into schemas.

1. `bronze`: Typically raw data, which is unaltered and as is from the source
2. `silver`: Cleaned or processed data, often transformed from raw data in Bronze
3. `gold`: Further enriched or aggregated data, ready for analysis or model training

TABLES

Feature tables:

- Tables with a primary key used to train models and serve features
- Here we illustrate how generic feature tables such as user and product features might appear under the "Gold" schema, given that such features could be shared across multiple use cases

4. `fraud_detection`: Consolidated schema for all machine learning assets relevant to the fraud detection use case, including (but not limited to):

MODELS

- Each registered model typically addresses some aspect of the schema-specific use case or project (in some cases there may be multiple models per use case)
- In our example we illustrate a single registered model for the fraud detection use case, called "fraud_clf" (fraud classifier)
- Every time a new MLflow model is registered to a given model, it will produce a new incremental model version
- Aliases are then assigned to specific model versions to mark them for deployment and to manage rollouts (e.g., "Champion," "Challenger")

VOLUMES

- Unstructured data (e.g., images, text) used to train models