TABLES

Feature tables:

- Feature tables created specifically for the fraud detection use case
- In the above example, we show a table called “offline_location_features” in development under the dev catalog. This table contains historic location-based features specific to the fraud detection use case.

Inference tables:

- Tables containing request-response data generated by Model Serving

Metric tables:

- Monitoring tables generated by Lakehouse Monitoring

FUNCTIONS

- Python UDFs used by models to compute features on demand at inference time
- In our example we illustrate a “compute_distance” function which computes real-time location-based features at inference time
- Note that functions can be used in many scenarios, and are not confined to just feature computation for models

For data engineering, we present a medallion architecture mapping to `bronze`, `silver` and `gold` schemas. This structure provides a step-by-step data transformation process, ensuring quality and consistency. While this naming convention is useful for example purposes, conceptual equivalents are often termed differently between organizations.

Model Serving

Incorporating real-time models into existing MLOps workflows necessitates a new set of considerations one must take into account. Collecting use case requirements such as service level agreements (SLAs) from your business stakeholders up front is essential in order to appropriately design a workflow that sufficiently tests a real-time model prior to deployment. There are two primary ways in which an MLOps workflow deploying a real-time model diverges from a MLOps workflow deploying batch inference pipelines:

1. **Pre-deployment testing**
   Ensuring system performance. A unique requirement for deploying real-time models is ensuring that model serving infrastructure is tested, in addition to regular pre-deployment testing (e.g., integration tests).

2. **Real-time model deployment**
   Ensuring model accuracy (or other ML performance metrics). Unlike batch inference pipelines, real-time models often require a paradigm of online model evaluation to ensure that the most accurate model is always serving predictions. This may involve updating a model on a more regular basis based on newly arriving data. As a result, this necessitates a different approach to deployment.

PRE-DEPLOYMENT TESTING

On top of the standard unit and integration tests that one would employ in a typical MLOps workflow, performance testing model serving infrastructure prior to deploying a real-time model is required in order to test how the model and serving infrastructure handles request traffic. Such testing could include:

- **Deployment readiness checks**
  These checks are conducted prior to creating or updating a Model Serving endpoint, to validate that configuration scripts are correctly specified, required dependencies are present, and the correct input data structure is defined, etc.

- **Load testing**
  Load testing involves conducting a comprehensive assessment of the real-time system's performance, stability and responsiveness under varying degrees of demand. The following are all components of load testing:

  #### LATENCY
  Ensure the model's inference and overall system response times meet predefined SLAs. Metrics like median latency and long tail (95th or 99th percentile) can capture both typical and worst-case response scenarios.

  #### THROUGHPUT
  Measure the application's capacity to handle queries over time, typically gauged in queries per second (QPS). Evaluations should consider varying load conditions to discern how performance fluctuates with demand shifts.

  #### STANDARD LOAD EVALUATION
  Analyze system behavior under expected request volumes, scaling from regular to anticipated peak levels. This not only gauges throughput but also examines metrics like response time and error rate.

  #### STRESS ASSESSMENT
  Deliberately overwhelm the system to observe its response to abnormally high demands, looking for graceful failure and effective recovery.

Note: The deployment patterns are not mutually exclusive. Organizations can, and often do, combine multiple patterns to suit requirements and risk profiles. For instance, one could start with a shadow deployment to evaluate a new model version without impacting the user experience, and based on findings, proceed with a gradual rollout.

REAL-TIME MODEL DEPLOYMENT

Given a newly trained ("Challenger") model and an existing ("Champion") model in production, there are a variety of approaches that can be taken for real-time deployments. Some common deployment patterns include:

- **A/B testing**
  Deploy multiple model versions concurrently and distribute the traffic among them to test and evaluate their performance. Based on predefined success criteria, such as accuracy or conversion rates, the best-performing model is then selected to handle all traffic.

  RELATION TO OTHER PATTERNS: A/B testing usually involves splitting traffic evenly or in some predefined ratio between model versions. Unlike gradual rollouts, where traffic is incrementally shifted based on performance metrics, A/B testing maintains its traffic splits until a decision is made.

- **Gradual rollout**
  This pattern begins by exposing a new model version to a small selected segment of request traffic (sometimes referred to as a canary deployment). Performance is closely monitored. If the new version meets defined success criteria, traffic to this model version is gradually increased. This allows for the benefits of continuous exposure to real traffic, while still having the safety net of scaling back if anomalies arise.

  RELATION TO OTHER PATTERNS: Unlike A/B testing, where traffic is set to predefined ratios, gradual rollout adjusts traffic based on model performance metrics. This is a more adaptive and responsive approach compared to other patterns.

- **Shadow deployment**
  In this pattern, a new model version runs alongside the existing version, but it does not actively serve traffic. Instead, the new version receives a copy of the incoming traffic and generates predictions, which can be compared to the current version for evaluation purposes without affecting the user experience.

  RELATION TO OTHER PATTERNS: Akin to a silent observer, where you can evaluate a new model version without affecting end users. This offers a risk-free comparison unlike a gradual rollout. Note that this involves performing additional work as multiple predictions are generated for the same input.

Implementing in Databricks

- **MODEL ALIASES**

  When implementing strategies like A/B testing or canary deployments, use model aliases to identify and manage different model versions. For example, using aliases you can easily switch traffic between a "Champion" model version and "Challenger" model version without needing extensive reconfigurations.

- **CONTROL ENDPOINT TRAFFIC**

  Databricks Model Serving provides the functionality to create a single Model Serving endpoint with two models and split endpoint traffic between those models. For instance, during A/B testing, you can set specific traffic percentages for each model version. Similarly, in rolling deployments, adjust these percentages as you phase in the new model.

- **LAKEHOUSE MONITORING WITH MODEL SERVING**

  With Lakehouse Monitoring you can set up monitoring on automatically captured inference tables from Model Serving endpoints. As such we can capture performance metrics of different model versions exposed to traffic. This can be particularly useful for any of the deployment patterns outlined above, where you need to closely monitor and compare predictions between different model versions.

While each deployment pattern outlined above has its strengths, the pattern you choose is ultimately dependent on the requirements of the specific use case and organization. With Databricks Model Serving, such patterns can be implemented easily, streamlining the deployment process for real-time ML models.

CHAPTER 5

Reference Architecture

In the reference architecture presented, we follow a deploy code workflow and assume a 1:1 mapping between environments and Unity Catalog. Hence, there are dev, staging and prod catalogs corresponding to assets produced in the development, staging and production environments, respectively. In this context an environment is the equivalent of a Databricks workspace.

Within each catalog we have the ability to manage both data and models. This architecture allows assets in the prod catalog to be accessed from the development environment, provided appropriate permissions are granted. Typically, this would involve granting read-only access to the prod catalog from the development environment, although it's important to note that not all organizations may allow this level of accessibility. Given this access, data scientists can develop ML code using production data in the development environment, where write access is restricted to only the dev catalog. Additionally, data scientists working within the development environment can load current production models residing in the prod catalog to compare against newly developed models. While our workflow involves registering and managing models within catalogs in Unity Catalog, each Databricks workspace has its own dedicated MLflow Tracking server to which metrics, parameters and model artifacts are logged.

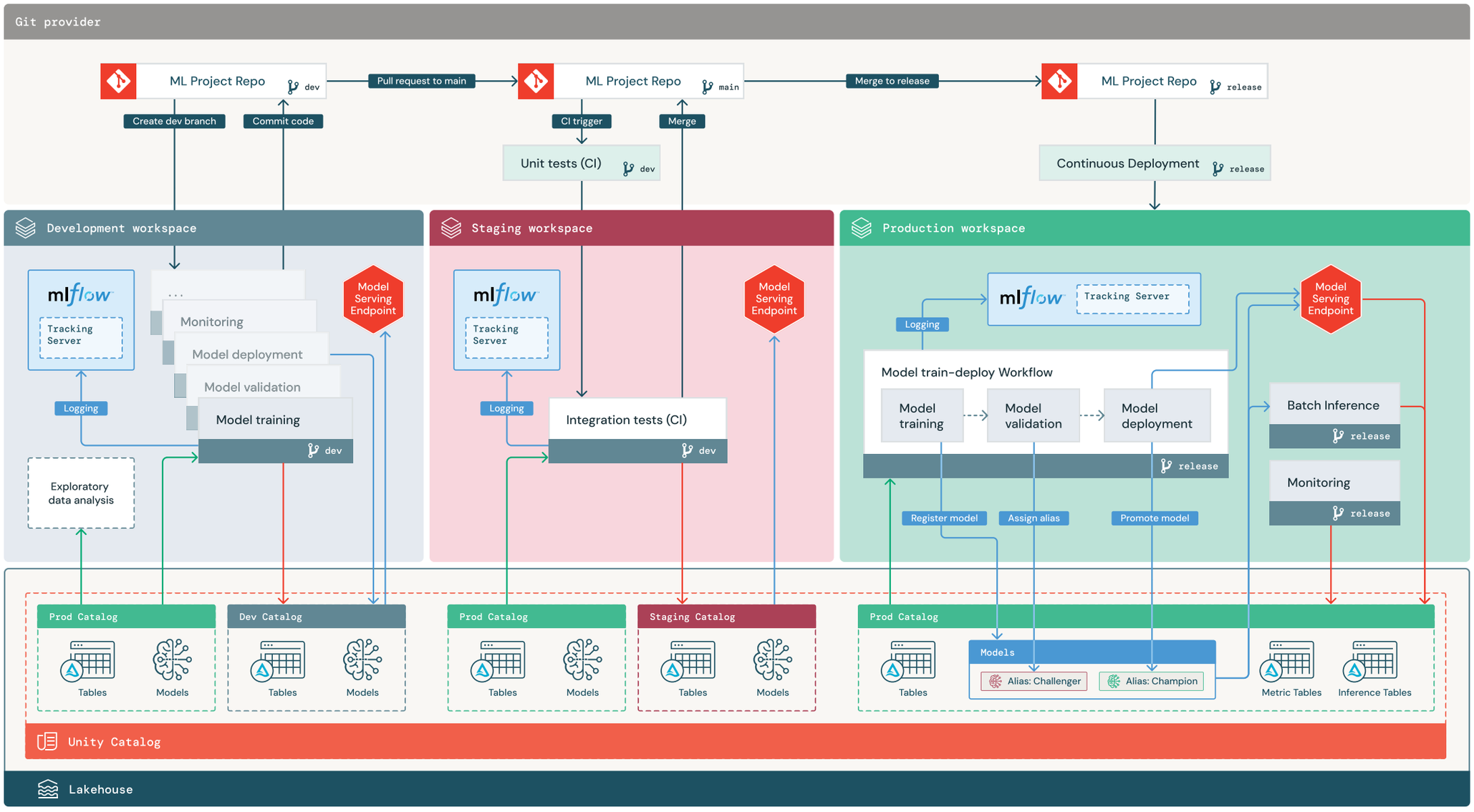
Note that environment terminology may vary across organizations and even teams. We present an architecture with development, staging and production environments; however, conceptually similar environments may have different naming conventions within your organization (e.g., QA/testing/pre-prod environment may be equivalent to staging).

Furthermore, just as environments can have different names, the branch naming and management strategies in Git can also vary. The Git workflow we present involves a dev branch merged into "main," and subsequently the main branch is merged into a "release" branch. However, in some cases teams may prefer working on "feature" branches corresponding to specific features or tasks in places of the dev branch. Similarly, the creation of a release branch might not always be necessary, with teams using tags to mark a particular commit as a release instead.

Thus, both environment names and CI/CD workflow depicted are intended to be used as guidance rather than to be prescriptive. The architecture and workflow outlined should be adapted to fit the unique needs and circumstances of your team and project.

Note: While the model promotion logic presented here is reliant on use of "Champion" and "Challenger" aliases, this naming convention is entirely adaptable. The flexibility of model aliases in Unity Catalog enables you to attribute aliases specific to your team's needs and terminologies.

Multi-environment view



Legend

Workflow · Job/Workflow task · CI/CD pipeline · Reads · Writes · MLflow API · Git repo · Git branch · Registered model

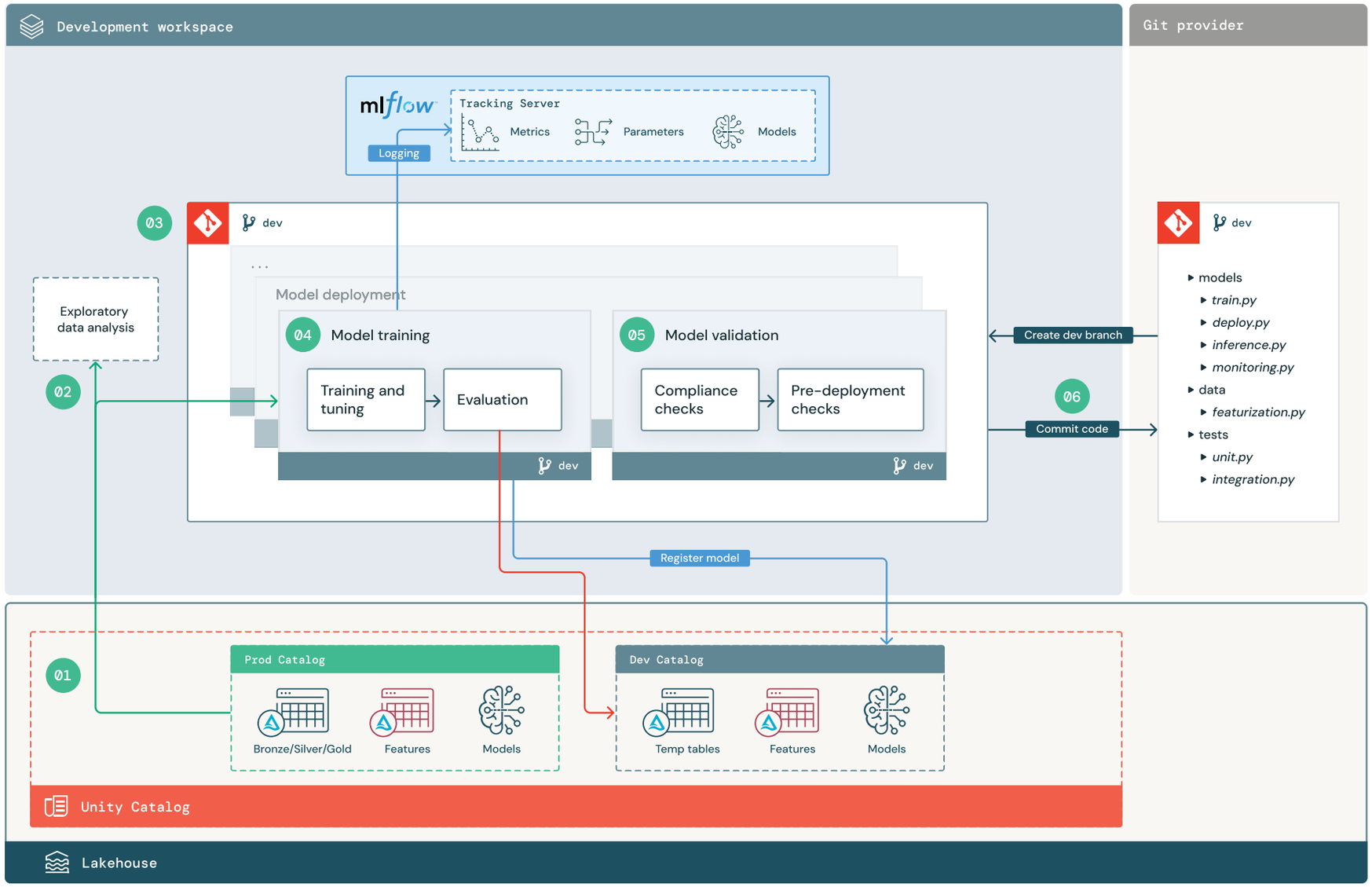
databricks

In the following sections we provide a detailed explanation of the precise steps in which code is moved across the three environments illustrated above. At a high level, we have the following steps:

1. **Development:** ML code is developed in the development environment, with code pushed to a dev (or feature) branch.
2. **Testing:** Upon making a pull request from the dev branch to the main branch, a CI trigger runs unit tests on the CI runner and integration tests in the staging environment.
3. **Merge code:** After successfully passing these tests, changes are merged from the dev branch to the main branch.
4. **Release code:** The release branch is cut from the main branch, and doing so deploys the project ML pipelines to the production environment.
5. **Model training and validation:** The model training pipeline ingests data from the prod catalog. Upon validating, the resulting model artifact is registered to the prod catalog. A "Challenger" alias is attached to the newly registered model version.
6. **Model deployment:** A model deployment pipeline evaluates the current "Champion" model versus "Challenger" model, with the best-performing model version taking the "Champion" alias after this evaluation.
7. **Model inference:** Model Serving or other inference pipelines load the "Champion" model to compute predictions. Predictions are logged to inference tables, which can be used to monitor the "Champion" model's performance.
8. **Monitoring:** Scheduled or continuous pipeline to refresh Lakehouse Monitoring metric tables. Inference tables are monitored to detect data or model drift. Databricks SQL dashboards are automatically created to display monitor metrics.

Development

Expanding on the above, we start by looking in detail at the development environment.



Legend

Workflow
Job/Workflow task
Pipeline setup
Reads
Writes
MLflow API
Git repo
Git branch
Registered model

databricks

Data

To support their development activities, data scientists will have read-write access to the dev catalog. This catalog is where temporary data and feature tables are written to from the development workspace. Additionally, this dev catalog is used to register any models created during code development to Unity Catalog.

Ideally, data scientists working within the development workspace will possess read-only access to production data in the prod catalog. This facilitates their ability to read production tables, as well as load models that have been registered to the prod catalog. In cases where it is not possible to grant read-only access to the prod catalog, a snapshot of production data may be written to the dev catalog to enable data scientists to develop and evaluate their project code.

Note that read access to inference and metric tables in the prod catalog will enable data scientists to examine current production model predictions and Lakehouse monitoring metrics.

Exploratory data analysis (EDA)

The data scientist's work typically begins with Exploratory Data Analysis (EDA) in the development environment. This is an interactive, iterative process — typically conducted in notebooks — to assess whether the available data has the potential to solve the business problem at hand. EDA is also where the data scientist will begin identifying data preparation and featurization steps for later model training. This ad hoc process is generally not part of a pipeline that will be deployed in other execution environments.

Within Databricks this EDA process can be accelerated with Databricks AutoML. AutoML not only generates baseline models given a data set, but also provides the underlying model training code in the form of a Python notebook. Notably for EDA, AutoML calculates summary statistics on the provided data set, creating a notebook for the data scientist to review and adapt.

Project code

This code repository contains all the pipelines, modules and ancillary project files involved in the ML solution. Development ("dev") branches are used to develop changes to existing pipelines or create new ones. Even during EDA and initial phases of a project, data scientists should develop within a repository to help with tracking changes and sharing code.

Model training development

Data scientists develop the model training pipeline in the development environment using Lakehouse tables from the dev or prod catalogs.

- **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 robust link between the model, its input data and the code used to generate it.

- **EVALUATION**

  Model quality is evaluated by testing on held-out data. The results of these tests are logged to the MLflow Tracking server. At this point it can be determined if a newly developed model outperforms that of the current model in production. Given sufficient permissions, any production model registered to the prod catalog can be loaded into the development workspace and compared against a newly trained model.

  If governance requires additional metrics or supplemental documentation about the model, this is the time to add that functionality to the code using MLflow Tracking. Model interpretations (e.g., plots produced by SHAP) and plain text descriptions are common, but defining the specifics for such governance requires input from business stakeholders or a data governance officer.

MODEL TRAINING OUTPUT

The model training pipeline produces an ML model artifact stored in the MLflow Tracking server. At this point the model artifact is tracked to the development environment MLflow Tracking server. However, when this pipeline is executed in either staging or production workspaces, the model is tracked to the respective MLflow Tracking servers of these workspaces.

Upon model training completion, the model is registered to Unity Catalog. When executed in the development environment, this model is registered to the dev catalog. Pipeline code is typically parameterized in such a manner that the model will be registered to the catalog corresponding to the environment the model training pipeline is executed in.

In the proposed architecture, we deploy a multitask Databricks Workflow in which the first task is the model training pipeline, followed by subsequent model validation, then model deployment tasks. As such, the model training task will yield a model URI (or path) which can be used by the subsequent model validation task. This model URI value can be passed into subsequent tasks using the taskValues subutility in Databricks Utilities.

Model validation and deployment development

In addition to the model training pipeline, other ancillary pipelines such as model validation and model deployment pipelines are developed in the development environment.

MODEL VALIDATION

This pipeline will take the model URI from the previous model training pipeline, load the model from Unity Catalog and apply validation checks.

The scope of validation checks on a newly trained model are typically context dependent. These checks can range from fundamental checks like asserting the model artifact's format and verifying the presence of required metadata for subsequent deployment and inference pipelines, to more complex checks, especially in highly regulated industries. The latter may involve predefined compliance checks and asserting that the model performance meets a certain threshold on selected data slices.