MLOps workflows on Azure Databricks

What is MLOps?

MLOps is a collection of automated procedures and processes for managing models, data, and code. ModelOps, DataOps, and DevOps are all combined.

Developing machine learning assets like code, data, and models involves several stages. Initially, there's a development phase with minimal access restrictions and limited testing. This progresses to an intermediate testing phase, and finally to a tightly controlled production stage. Databricks provides a unified platform to manage these assets seamlessly. This allows you to build both data and machine learning applications in one place, minimizing the risks and delays that come with transferring data between different systems.

Create a separate environment for each stage

An execution environment is where models and data are created or used by code. It includes compute instances, their runtimes, libraries, and automated jobs.

Databricks suggests setting up separate environments for each stage of ML code and model development, with clear transitions between these stages. This document outlines a workflow that follows this process, using the common names for each stage:

- Development
- Staging
- Production

You can also use other configurations to meet your organization's specific needs.

Access control and versioning

Access control and versioning are crucial parts of any software operations process. Databricks suggests the following:

- **Use Git for version control**. Store pipelines and code in Git to manage versions effectively. Transitioning ML logic between stages can be seen as moving code from the development branch to the staging branch and finally to the release branch. Utilize Databricks Git folders to connect with your Git provider and sync notebooks and source code with Databricks

workspaces. Databricks also offers extra tools for Git integration and version control; refer to Developer tools and guidance for more details.

- **Store data in a lakehouse architecture using Delta tables.** Keep your data in a lakehouse setup in your cloud account, with both raw data and feature tables stored as Delta tables. Implement access controls to manage who can read and modify the data.
- **Manage model development with MLflow.** Use MLflow to track your model development process, saving code snapshots, model parameters, metrics, and other metadata.
- **Use Models in Unity Catalog to manage the model lifecycle.** Leverage Models in Unity Catalog to handle model versioning, governance, and deployment status.

Deploy code, not models

In most cases, Databricks recommends promoting code, rather than models, during the ML development process from one environment to the next. This approach ensures that all code in the ML development cycle undergoes consistent code review and integration testing processes. It also guarantees that the production version of the model is trained on production code. For a deeper dive into the available options and trade-offs, refer to the Model deployment patterns section.

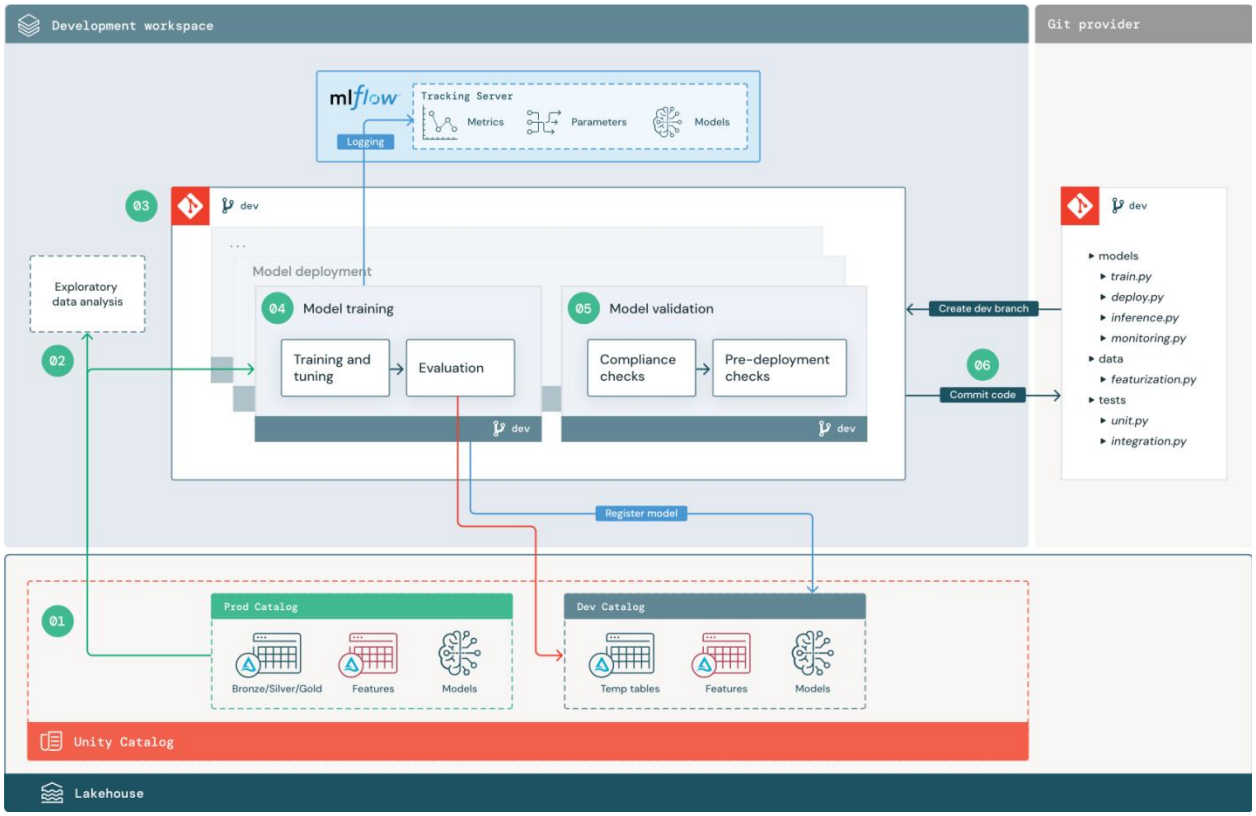
Recommended MLOps workflow

The following sections outline a typical MLOps workflow, encompassing the development, staging, and production stages.

This section uses the terms "data scientist" and "ML engineer" as archetypal personas; specific roles and responsibilities within the MLOps workflow may vary between teams and organizations.

Development stage

The focus of the development stage is experimentation. Data scientists work on developing features and models, conducting experiments to optimize model performance. The output of the development process is ML pipeline code, which can include feature computation, model training, inference, and monitoring.



The numbered steps align with the numbers depicted in the diagram.

1. Data sources

In the development environment, which is managed through the dev catalog in Unity Catalog, data scientists have read-write access. They use this access to create temporary data and feature tables in the development workspace. Models that are created during the development stage are registered in the dev catalog.

It would be ideal if data scientists working in the development workspace could also have read-only access to production data stored in the prod catalog. This access would enable them to examine current production model predictions and performance by accessing inference tables and metric tables in the prod catalog. Moreover, data scientists should have the capability to load production models for experimentation and analysis.

If it's not feasible to grant read-only access to the prod catalog, another approach is to write a snapshot of production data to the dev catalog. This enables data scientists to develop and evaluate project code effectively.

2. Exploratory data analysis (EDA)

Data scientists interactively explore and analyze data using notebooks in an iterative process. The objective is to evaluate whether the available data can effectively address the business problem. During this step, data scientists start identifying data preparation and featurization steps for model training. This ad hoc process typically isn't part of a pipeline that will be deployed in other execution

Databricks AutoML speeds up this process by generating baseline models for a dataset. AutoML conducts and logs a series of trials and provides a Python notebook for each trial run, allowing you to review, reproduce, and customize the code as needed. AutoML also computes summary statistics on your dataset and saves this information in a notebook that you can examine.

3. Code

The code repository holds all 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. From exploratory data analysis (EDA) and the initial phases of a project, data scientists should collaborate in a repository to share code and track changes.

4. Train model (development)

In the development environment, data scientists craft the model training pipeline using tables sourced from the dev or prod catalogs.

This pipeline consists of two tasks:

1. **Training and tuning:** During the training process, model parameters, metrics, and artifacts are logged to the MLflow Tracking server. Once hyperparameters are tuned and training is complete, the final model artifact is logged to the tracking server. This records a link between the model, the input data it was trained on, and the code used to generate it.
2. **Evaluation:** Evaluation involves testing the model on held-out data to assess its quality. The outcomes of these tests are logged to the MLflow Tracking server. The objective of evaluation is to ascertain whether the newly developed model outperforms the current production model. If permitted, any production model listed in the prod catalog can be loaded into the development workspace and juxtaposed with the

If your organization has governance requirements that necessitate additional model information, you can store it using MLflow tracking. This typically includes artifacts like plain-text descriptions and model interpretations such as plots generated by SHAP. Specific governance requirements may be specified by a data governance officer or business stakeholders.

The result of the model training pipeline is an ML model artifact stored in the MLflow Tracking server specific to the development environment. If the pipeline runs in the staging or production workspace, the model artifact is stored in the MLflow Tracking server designated for that workspace.

Once the model training is complete, register the model in the Unity Catalog. Configure your pipeline code to register the model in the catalog that corresponds to the environment where the model pipeline was executed; in this case, the dev catalog.

In the recommended architecture, you deploy a multitask Databricks workflow where the first task is the model training pipeline, followed by model validation and model deployment tasks. The model training task produces a model URI that the model validation task can utilize. You can utilize task values to pass this URI to the model.

5. Validate and deploy model (development)

Alongside the model training pipeline, other pipelines, such as model validation and model deployment pipelines, are also developed in the development environment.

- **Model validation:** Model validation involves taking the model URI from the model training pipeline, loading the model from Unity Catalog, and conducting validation checks.

  Validation checks vary based on the context. They may involve basic checks like confirming the format and required metadata, as well as more intricate checks needed for highly regulated industries. Examples include predefined compliance checks and confirming model performance on selected data slices.

  The main purpose of the model validation pipeline is to decide if a model is ready to proceed to the deployment phase. If the model successfully passes the pre-deployment checks, it can be assigned the "Challenger" alias in Unity Catalog. However, if the checks fail, the process terminates. You can set up your workflow to notify users in case of a validation failure.

- **Model deployment:** Model deployment involves the model deployment pipeline, which usually 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 configure any necessary inference infrastructure, such as Model Serving endpoints.

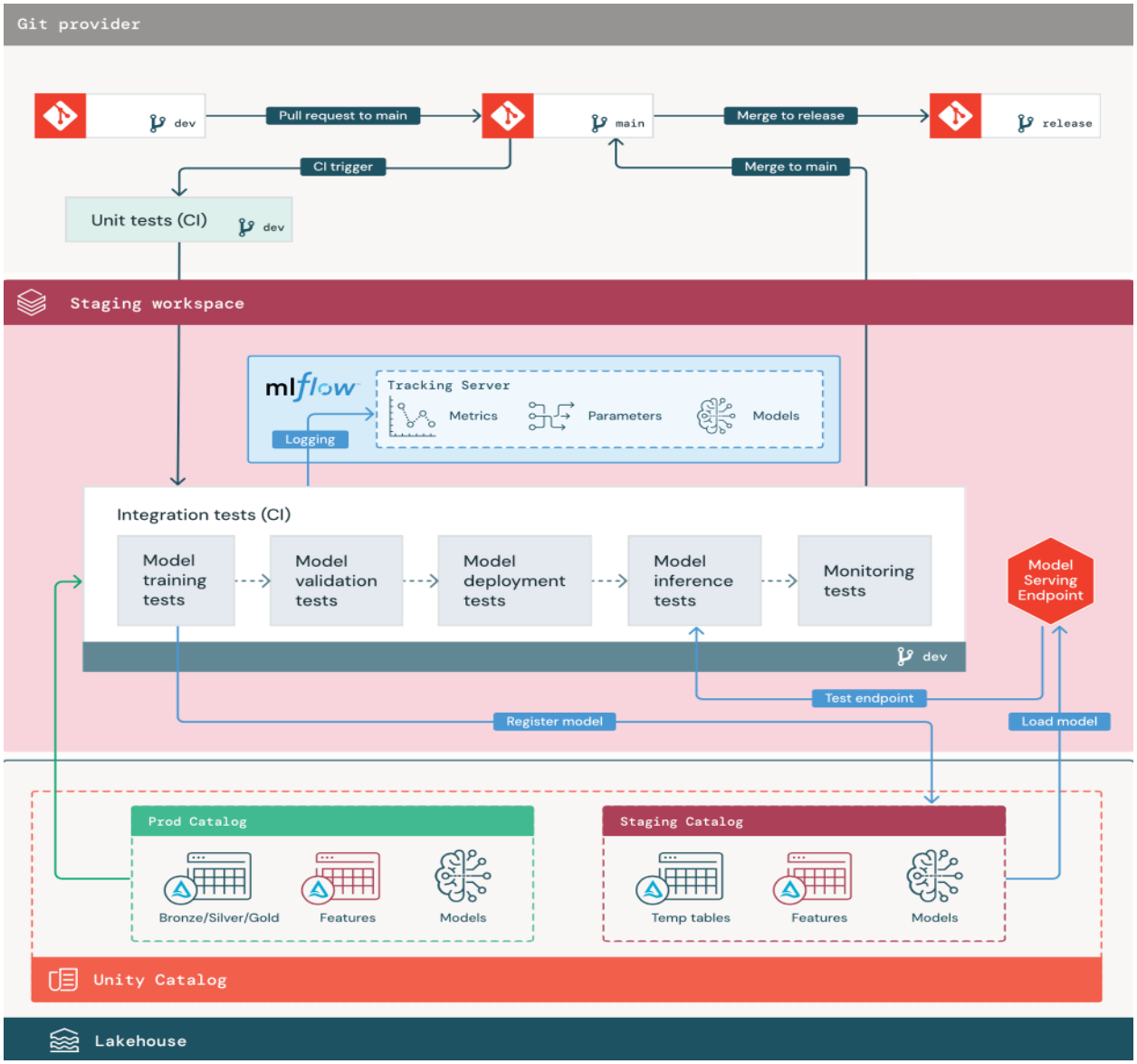
6. Commit code

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 primary focus of this stage is to thoroughly test the ML pipeline code to ensure its readiness for production. This includes testing all components of the ML pipeline, such as model training code, feature engineering pipelines, inference code, and other relevant aspects.

ML engineers establish a CI pipeline to execute unit and integration tests during this stage. The output of the staging process is a release branch, which triggers the CI/CD system to initiate the production stage.



1. Data

The staging environment should possess its own catalog in Unity Catalog dedicated to testing ML pipelines and registering models. This catalog, depicted as the "staging" catalog in the diagram, typically stores temporary assets that are retained only until testing is finished. Additionally, the development environment might need access to the staging catalog for debugging purposes.

2. Merge code

Data scientists create the model training pipeline in the development environment, using tables from either the development or production catalogs.

- **Pull request:** The deployment process initiates when a pull request is opened against the main branch of the project in source control.
- **Unit tests (CI):** When a pull request is made, the source code is automatically built and unit tests are triggered. If the unit tests fail, the pull request is declined.

Unit tests are a crucial part of the software development process. They are continuously executed and integrated into 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.

3. Integration tests (CI)

The CI process proceeds to execute integration tests. Integration tests validate all pipelines, including feature engineering, model training, inference, and monitoring, to ensure they operate correctly together. It's important that the staging environment closely resembles the production environment.

For ML applications with real-time inference, it's necessary to create and test serving infrastructure in the staging environment. This involves initiating the model deployment pipeline, which sets up a serving endpoint in the staging environment and deploys a model.

To expedite the execution of integration tests, some steps can prioritize speed or cost over the fidelity of testing. For instance, if training models is expensive or time-consuming, you might use small subsets of data or reduce the number of training iterations. Similarly, for model serving, depending on production needs, you might conduct full-scale load testing during integration tests, or simply test small batch jobs or requests through a temporary endpoint.

4. Merge to staging branch

If all tests are successful, the new code gets merged into the main branch of the project. However, if any tests fail, the CI/CD system should notify users and provide results on the pull request.

Scheduling periodic integration tests on the main branch is advisable, especially if the branch receives frequent updates from multiple users through concurrent pull requests.

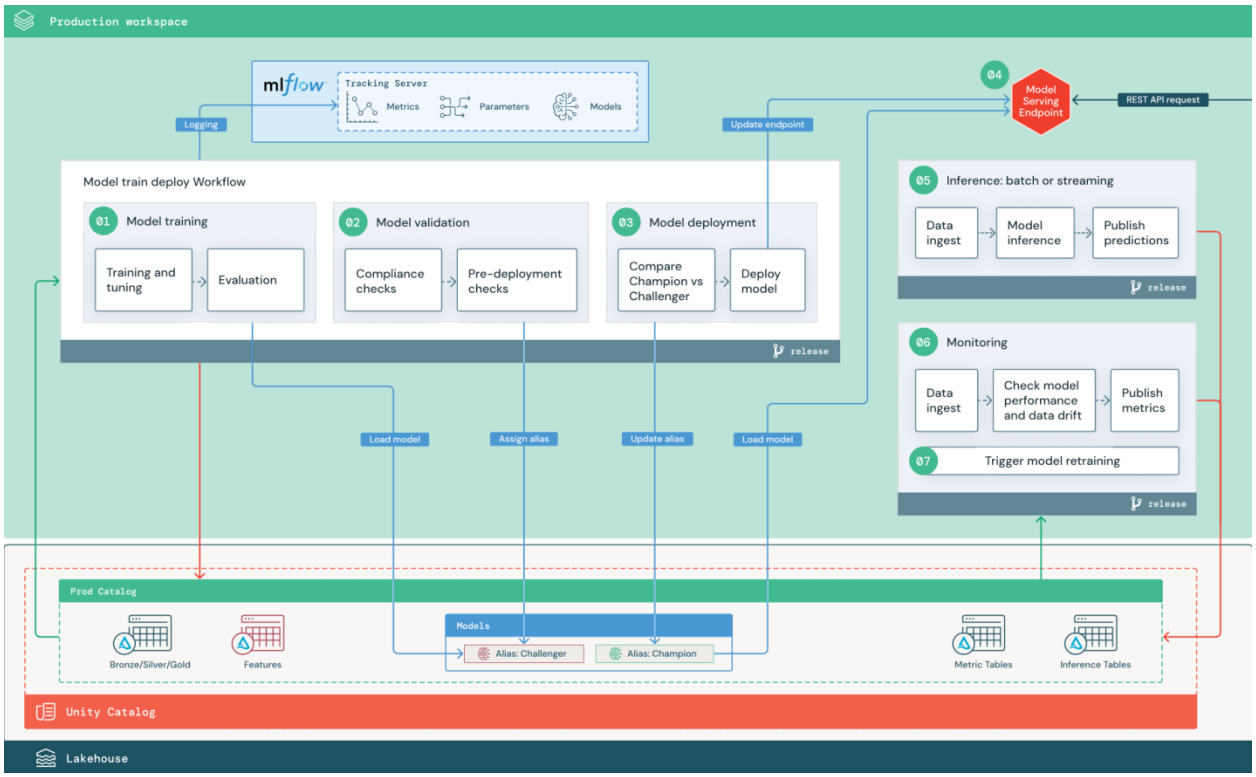
5. Create a release branch

Once the CI tests have been successfully completed and the changes from the development branch are merged into the main branch, the ML engineer generates a release branch. This action triggers the CI/CD system to update production jobs.

Production stage

ML engineers are responsible for managing the production environment where ML pipelines are deployed and executed. These pipelines handle tasks such as triggering model training, validating and deploying new model versions, publishing predictions to downstream tables or applications, and monitoring the entire process to prevent performance degradation and instability.

Data scientists usually do not have write or compute access in the production environment. However, it is crucial that they have visibility into test results, logs, model artifacts, production pipeline status, and monitoring tables. This visibility enables them to identify and diagnose issues in production and compare the performance of new models with those currently in production. To facilitate this, you can provide data scientists with read-only access to assets in the production catalog.



The numbered steps correspond to the numbers shown in the diagram.

1. Train model

This pipeline can be initiated by code alterations or by automated retraining jobs. During this phase, tables from the production catalog are utilized for the subsequent steps.

- **Training and tuning:** Throughout the training process, logs are stored in the MLflow Tracking server in the production environment. These logs contain model metrics, parameters, tags, and the model itself. If feature tables are used, the model is logged to MLflow using the Databricks Feature Store client. This client packages the model along with feature lookup information that is crucial during inference.

  During development, data scientists often test various algorithms and hyperparameters. However, in the production training code, it's typical to focus only on the top-performing options. Restricting tuning in this manner saves time and can minimize variance from tuning in automated retraining.

  If data scientists have read-only access to the production catalog, they can identify the optimal set of hyperparameters for a model. Subsequently, the model training pipeline deployed in production can use this selected set of hyperparameters, typically included in the pipeline as a configuration file.

- **Evaluation**: Model quality is assessed by testing on held-out production data. The outcomes of these tests are logged to the MLflow tracking server. This step employs the evaluation metrics defined by data scientists during the development stage, which may include custom code.
- **Registering the model:** After model training is finished, the model artifact is saved as a registered model version at the designated model path in the production catalog in Unity Catalog. The model training task produces a model URI that the model validation task can utilize, and task values can be used to pass this URI to the model.

2. Validate model

This pipeline utilizes the model URI from Step 1 to load the model from Unity Catalog. It then performs a series of validation checks. These checks vary based on your organization and use case, and may include basic format and metadata validations, performance evaluations on selected data slices, and compliance with organizational requirements such as tag or documentation compliance checks.

If the model passes all validation checks, you can label the model version as "Challenger" in Unity Catalog. However, if the model fails any validation checks, the process stops and users can receive automated notifications. Tags can be utilized to attach key-value attributes depending on the outcome of these validation checks. For example, you could create a tag called "model_validation_status" and set the value to "PENDING" while the tests are running. After the pipeline is complete, you can update it to "PASSED" or "FAILED" accordingly.

Since the model is registered in Unity Catalog, data scientists in the development environment can load this model version from the production catalog to investigate if the model fails validation. Regardless of the outcome, the results are recorded to the registered model in the production catalog using annotations to the model version.

3. Deploy model

Similar to the validation pipeline, the model deployment pipeline is tailored to your organization and use case. This section assumes that you have designated the newly validated model with the “Challenger” alias, and the current production model with the “Champion” alias.

Before deploying the new model, the first step is to verify that it performs at least as well as the current production model.

- **Compare the “CHALLENGER” to the “CHAMPION” model:** This comparison can be conducted offline or online. An offline comparison evaluates both models against a held-out

dataset 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.

Databricks Model Serving and Databricks Lakehouse Monitoring enable automatic collection and monitoring of inference tables containing request and response data for an endpoint.

In the absence of an existing "Champion" model, you could compare the "Challenger" model against a business heuristic or another threshold to establish a baseline.

The described process is entirely automated. However, if manual approval steps are necessary, you can establish them using workflow notifications or CI/CD callbacks from the model deployment pipeline.

- **Deploy model:** To deploy the model, you can set up batch or streaming inference pipelines to use the model with the "Champion" alias. For real-time applications, you need to establish the infrastructure to deploy the model as a REST API endpoint. You can create and manage this endpoint using Databricks Model Serving. If an endpoint is already in use for the current model, you can update it with the new model. Databricks Model Serving ensures a zero-downtime update by maintaining the existing configuration until the new one is ready.

4. Model Serving

When setting up a Model Serving endpoint, you simply provide the name of the model in Unity Catalog and the version to serve. If the model version used features from tables in Unity Catalog during training, it stores the dependencies for these features and functions. At inference time, Model Serving automatically utilizes this dependency graph to retrieve features from appropriate online stores. This method can also be employed to apply functions for data preprocessing or to compute on-demand features during model scoring.

You have the flexibility to create a single endpoint with multiple models and allocate the endpoint traffic split between these models. This allows you to conduct online comparisons between "Champion" and "Challenger" models.

5. Inference: batch or streaming

For the inference pipeline, it starts by fetching the latest data from the production catalog. Then, it executes functions to compute on-demand features, loads the "Champion" model, scores the data, and returns predictions. Generally, batch or streaming inference is the most cost-effective option for use cases that prioritize higher throughput with slightly higher latency.

In situations where low-latency predictions are needed, but they can be computed offline, batch predictions can be published to an online key-value store like 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. This approach ensures that the model deployment step remains independent from inference pipelines.

Batch jobs typically publish predictions to tables in the production catalog, flat files, or via a JDBC connection. On the other hand, streaming jobs usually publish predictions to Unity Catalog tables or message queues such as Apache Kafka.

6. Lakehouse Monitoring

Lakehouse Monitoring keeps an eye on statistical properties, such as data drift and model performance, for both input data and model predictions. Alerts can be created based on these metrics or they can be displayed on dashboards.

- **Data ingestion:** This pipeline reads logs from batch, streaming, or online inference processes.
- **Check accuracy and data drift:** This pipeline calculates metrics concerning the input data, the model's predictions, and the performance of the infrastructure. During development, data scientists define data and model metrics, while ML engineers specify infrastructure metrics. Additionally, custom metrics can be configured using Lakehouse Monitoring.
- **Publish metrics and set up alerts:** The pipeline logs data to tables in the production catalog for analysis and reporting. Ensure these tables are accessible from the development environment so data scientists can review them. Use Databricks SQL to create dashboards for monitoring model performance, and configure the monitoring job or dashboard tool to send notifications when a metric goes beyond a set threshold.
- **Trigger model retraining:** If monitoring metrics show performance issues or changes in the input data, a new model version might be needed. Set up SQL alerts to notify data scientists when this occurs.

7. Retraining

This setup allows for automatic retraining using the existing model training pipeline. Databricks suggests starting with scheduled, periodic retraining and transitioning to triggered retraining as necessary.

- **Scheduled:** If new data becomes available regularly, you can set up a scheduled job to run the model training code on the most recent data.
- **Triggered:** When the monitoring pipeline detects model performance issues, it can trigger retraining. For example, if the incoming data distribution changes significantly or the model's performance declines, automatic retraining and redeployment can improve model performance with minimal human intervention. You can set up a SQL alert to identify anomalous metrics, such as data drift or model quality falling below a threshold. This alert can then trigger the training workflow via a webhook.