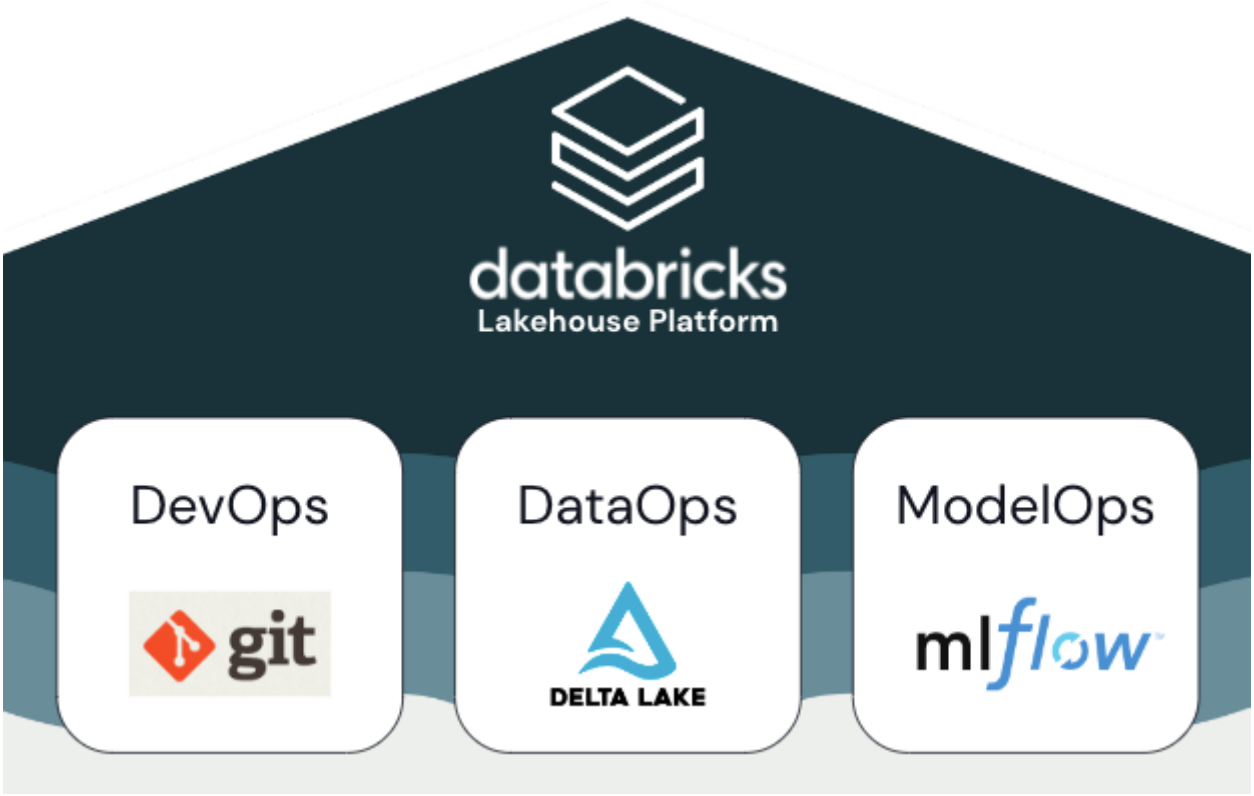
MLOps workflow on Azure Databricks - Azure Databricks | Microsoft Learn

1 June 2023 Skip to main content . In this article

This article describes how you can use MLOps on the Databricks Lakehouse platform to optimize the performance and long-term efficiency of your machine learning (ML) systems. It includes general recommendations for an MLOps architecture and describes a generalized workflow using the Databricks Lakehouse platform that you can use as a model for your ML development-to-production process.

What is MLOps?

MLOps is a set of processes and automated steps to manage code, data, and models. 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 Lakehouse platform lets you manage 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

This section includes some general recommendations for MLOps on Databricks with links for more information.

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.

Databricks recommends creating separate environments for the different stages of ML code and model development with clearly defined transitions between stages. The workflow described in this article follows this process, using the common names for the stages:

- Development
- Staging
- Production

Other configurations can also be used to meet the specific needs of your organization.

Access control and versioning

Access control and versioning are key components of any software operations process. Databricks recommends the following:

- **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 Repos to integrate with your Git provider and sync notebooks and source code with Databricks workspaces. Databricks also provides additional tools for Git integration and version control; see Developer tools and guidance.
- **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 models and model development with MLflow.** You can use MLflow to track the model development process and save code snapshots, model parameters, metrics, and other metadata. Use the Model Registry to manage model versioning and deployment status. The Model Registry provides webhooks and an API so you can integrate with CD systems, and also handles access control for models.

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. For a more detailed discussion of the options and trade-offs, see Model deployment patterns.

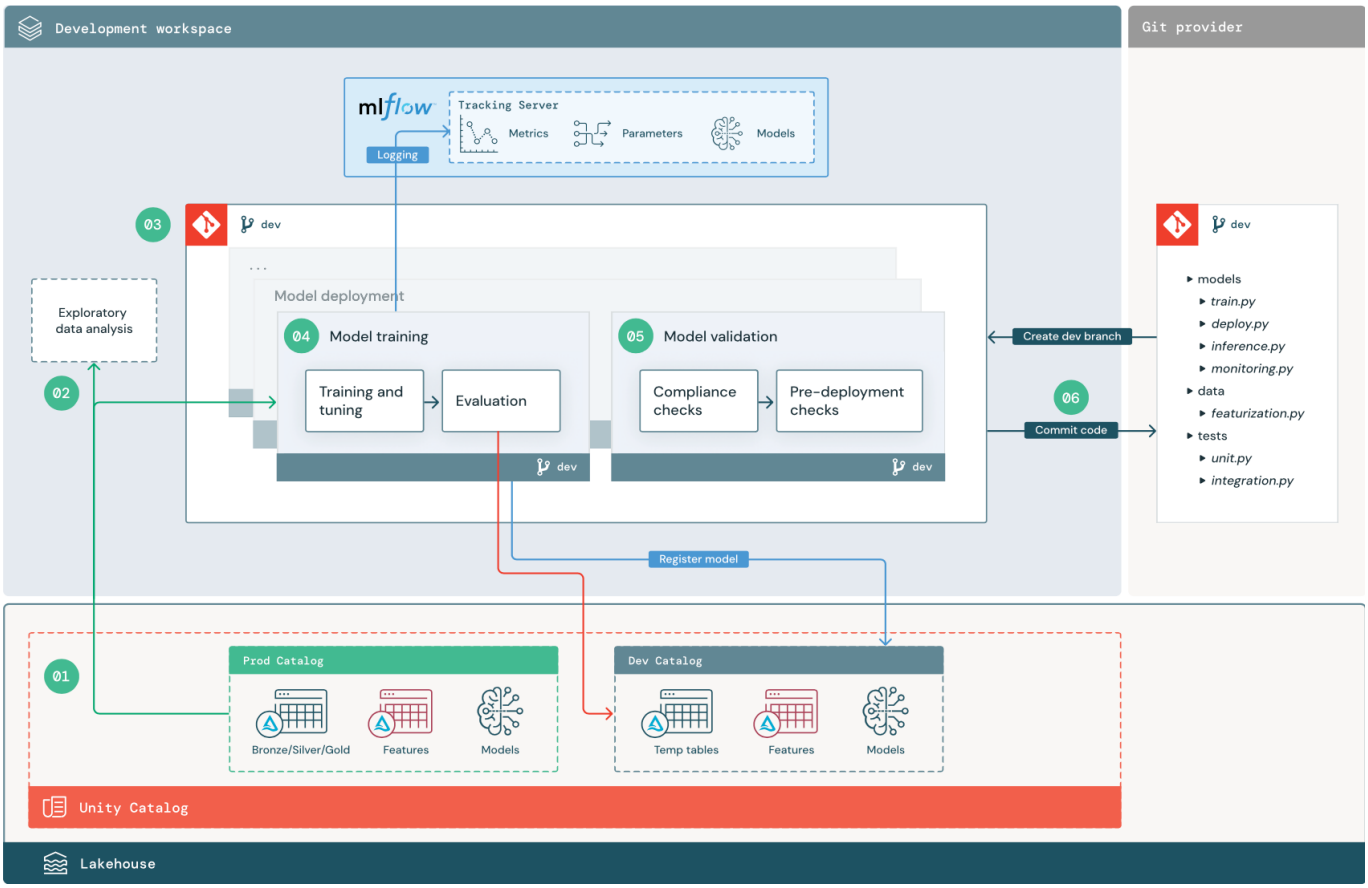
Recommended MLOps workflow

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 workflow will vary between teams and organizations.

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.



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

1. Data sources

Data scientists working in the dev environment often have read-only access to production data. In some cases to meet data governance requirements, the dev environment may have access only to a mirror or redacted version of production data. Data scientists also have read-write access to a separate dev storage environment to develop and experiment with new features and other data tables.

2. Exploratory data analysis (EDA)

Data scientists explore and analyze data in an interactive, iterative process using notebooks, visualizations, and Databricks SQL.

This ad hoc process is generally not part of a pipeline which will be deployed in other execution environments.

3. Code

All of the code for the ML system is stored in a code repository. Data scientists create new or updated pipelines in a development branch of the Git project. The code can be developed inside or outside of Azure Databricks and synced with the Azure Databricks workspace using Databricks Repos.

4. Update feature tables

The model development pipeline reads from both raw data tables and existing feature tables, and writes to tables in the Feature Store. This pipeline includes 2 tasks:

- **Data preparation.** Check for data quality issues.

- **Create or update feature tables.** Data scientists develop or update code to create features. These pipelines can read from the Feature Store and other Lakehouse tables and write to feature tables in the dev storage environment. Data scientists then use these dev feature tables to create prototype models. When the code is promoted to production, these changes update the production feature tables.

  Feature pipelines can be managed separately from other ML pipelines, especially if they are owned by different teams.

5. Train model

Data scientists develop model training and other pipelines either on read-only production data or on non-production data. The pipelines can use feature tables in either the dev or prod environments.

This pipeline includes 2 tasks:

- **Training and tuning.** The model training process reads features from the feature store and silver- or gold-level Lakehouse tables, and it logs model parameters, metrics, and artifacts to the MLflow tracking server.

  When training and hyperparameter tuning are complete, the data scientist saves the final model artifact to the tracking server. This records a link between the model, its input data, and the code used to generate it.

  When this training pipeline is run in staging or production, ML engineers (or their CI/CD code) can load the model by using the model URI (or path) and then push the model to the Model Registry for management and testing.

- **Evaluation.** Evaluate model quality by testing on held-out data. The results of these tests are logged to the MLflow tracking server.

  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 like those produced by SHAP or LIME.

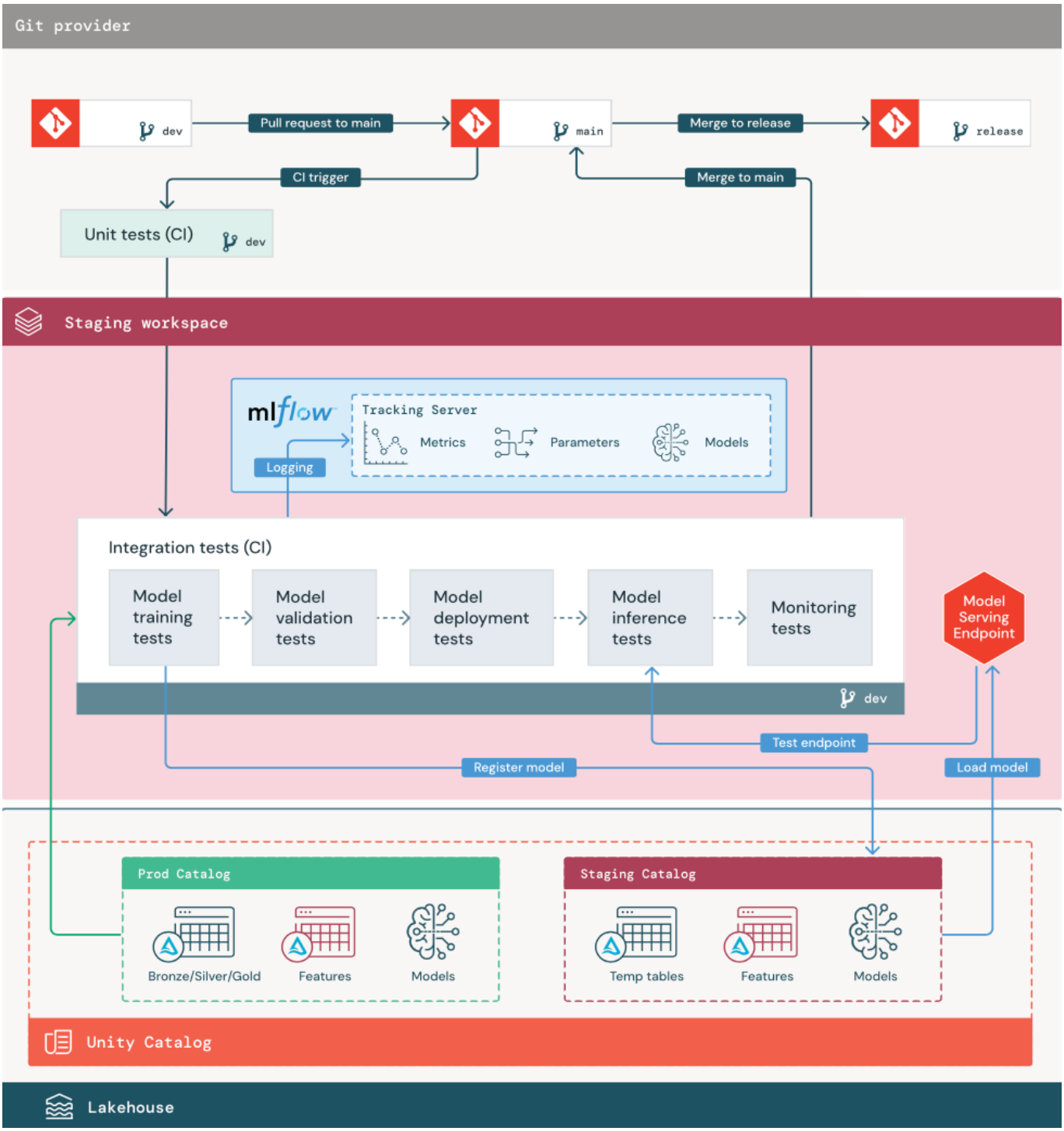
6. Commit code

After developing code for feature engineering, training, inference, 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.



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

The staging environment can have its own storage area for testing feature tables and ML pipelines. This storage is generally temporary and only retained until testing is complete. The development environment may also require access to this data storage for debugging purposes.

1. Merge request

The deployment process begins when an ML engineer creates a merge request to the staging branch (usually the “main” branch) in source control. The merge request triggers a continuous integration (CI) process.

2. Unit tests

The CI process automatically builds the source code and triggers unit tests. If the tests fail, the merge request is rejected. Unit tests do not interact with data or other services.

3. 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.

To reduce the time required to run integration tests, model training steps can trade off between fidelity of testing and speed. For example, you might use small subsets of data or run fewer training iterations. Depending on the intended use of the model, you may choose to do full-scale load testing at this point.

After the integration tests pass on the staging branch, you can promote the code to production.

4. Merge to staging branch

If the tests pass, the code can be merged to the staging branch. If tests fail, the CI/CD system should notify users and post results on the merge (pull) request.

You can schedule periodic integration tests on the staging branch. This is a good idea if the branch is updated frequently by different users.

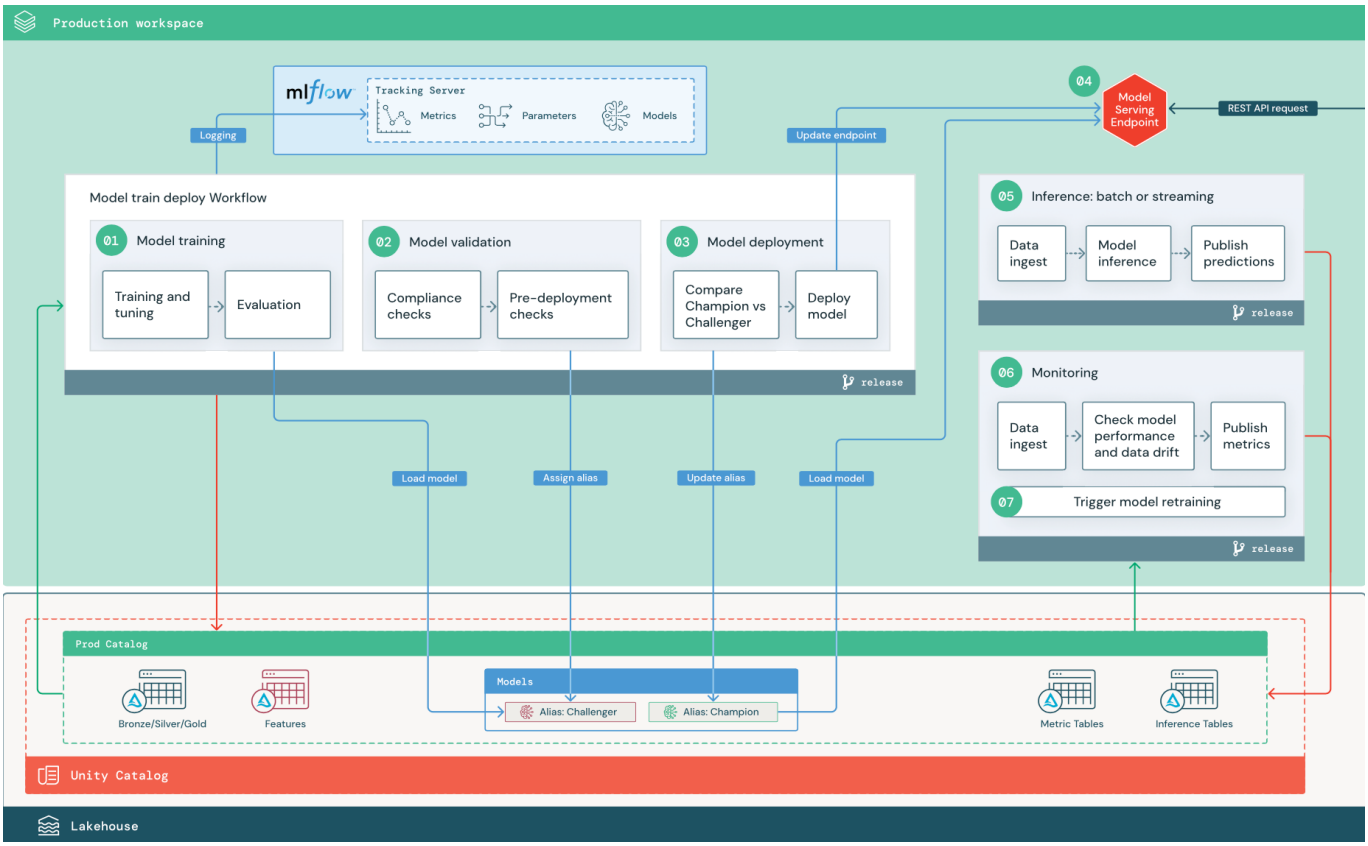
5. Create a release branch

When the code is ready to deploy to production, 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. These pipelines compute fresh feature values, train and test 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, and production pipeline status, to allow them to identify and diagnose problems in production.



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

1. Update feature tables

As new production data becomes available, this pipeline ingests it and updates the production feature store tables. This pipeline can be run as a batch or streaming job and can be scheduled, triggered, or run continuously.

2. Train model

Train the production version of the model on the full production data and register it with the MLflow Model Registry. This pipeline can be triggered by code changes or by automated retraining jobs.

This pipeline includes 2 tasks:

- **Training and tuning.** As in the development stage, autologging saves a record of the training process to the MLflow tracking server. This includes model metrics, parameters, tags, and the model itself.

  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.

- **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.

When model training is complete, register the model artifact in the MLflow Model Registry for the production environment. The initial Model Registry stage is "None".

3. Continuous deployment (CD)

A CD process takes new models (in Model Registry “stage=None”), tests them (transitioning through “stage=Staging”), and if successful deploys them (promoting them to “stage=Production”). CD can be implemented using Model Registry webhooks or your own CD system.

This pipeline includes 3 tasks:

- **Compliance check.** These tests load the model from the Model Registry, perform any compliance checks required by your organization (for example, tags or documentation) and approve or reject the request based on test results. If compliance checks require human expertise, this automated step can compute statistics or visualizations for manual review. Regardless of the outcome, record results for the model version to the Model Registry using metadata in tags and comments in descriptions.

  You can use the MLflow UI to manage stage transitions and transition requests manually or use MLflow APIs and webhooks to automate them. If the model passes the compliance checks, then the transition request is approved and the model is promoted to ‘stage=Staging’. If the model fails, the transition request is rejected and the model is moved to ‘stage=Archived’ in the Model Registry.

- **Compare staging to production.** To prevent performance degradation, you should compare the performance of a model promoted to Staging to the current Production version. The comparison metrics and methods depend on the use case and can include canary deployments, A/B testing, or other methods. Results of comparison testing should be saved to metrics tables in the Lakehouse.

  If this is the first deployment and there is no Production version yet, you can compare the Staging version to a business heuristic or other threshold as a baseline.

- **Request model transition to Production.** If the candidate model passes the comparison tests, you can request to transition it in the Model Registry to ‘stage=Production’. You can do this manually using the MLflow UI or automatically using the MLflow API and webhooks. It is also a good idea to consider requiring human approval at this point. This is the final step before a model is released to production and integrated into existing business processes. You can include a human review to verify compliance checks, performance comparisons, and any other checks that are difficult to automate.

4. Online serving (REST APIs)

For low-latency use cases, you can use MLflow to deploy the model for online serving. Options include Databricks Model Serving, cloud provider serving endpoints, or custom serving applications.

The serving system loads the Production model version from the Model Registry. For each request, it fetches features from an online Feature Store, scores the data, and returns predictions. You can log requests and predictions using the serving system, the data transport layer, or the model.

5. Inference: batch or streaming

For batch or streaming inference jobs, the pipeline reads the latest data from the Feature Store, loads the Production model version from the Model Registry, scores the data, and returns predictions. Batch or streaming inference is generally the most cost-effective option for higher throughput, higher latency use cases.

Batch jobs typically publish predictions to Lakehouse tables, over a JDBC connection, or to flat files. Streaming jobs typically publish predictions to Lakehouse tables or to message queues like Apache Kafka.

6. Monitoring

You should monitor input data and model predictions for statistical properties (such as data drift and model performance) and for computational performance (such as errors and throughput). You can create alerts based on these metrics or publish them in dashboards.

Regardless of deployment mode, you can log the model's input queries and predictions to Delta tables. You can create jobs to monitor data and model drift, and you can use Databricks SQL to display status on dashboards and send alerts. Data scientists can be granted access to logs and metrics in the development environment to investigate production issues.

This pipeline includes 3 tasks:

- **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.
- **Publish metrics.** The pipeline writes to Lakehouse tables for analysis and reporting. 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.

7. Trigger model retraining

You can create a scheduled job to retrain a model with the latest data, or you can set up a monitor to trigger retraining when it detects drift in the data or the model. If the model monitoring metrics indicate performance issues, the data scientist may need to return to the development environment and develop a new model version.

Note

Fully automated model retraining is difficult to get right, as it may not be obvious how to fix a problem detected by model monitoring. For example, model performance problems caused by observed data drift might be fixed by retraining the model on newer data, or might require additional (manual) feature development work to encode a new signal in the data.

- 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.
- If the monitoring pipeline can identify model performance issues and send alerts, you can configure it to automatically trigger retraining. Automatic retraining and redeployment can improve model performance with minimal human intervention if the pipeline can detect situations such as a change in the distribution of incoming data or a degradation in model performance.

Feedback

Submit and view feedback for

Additional resources

In this article