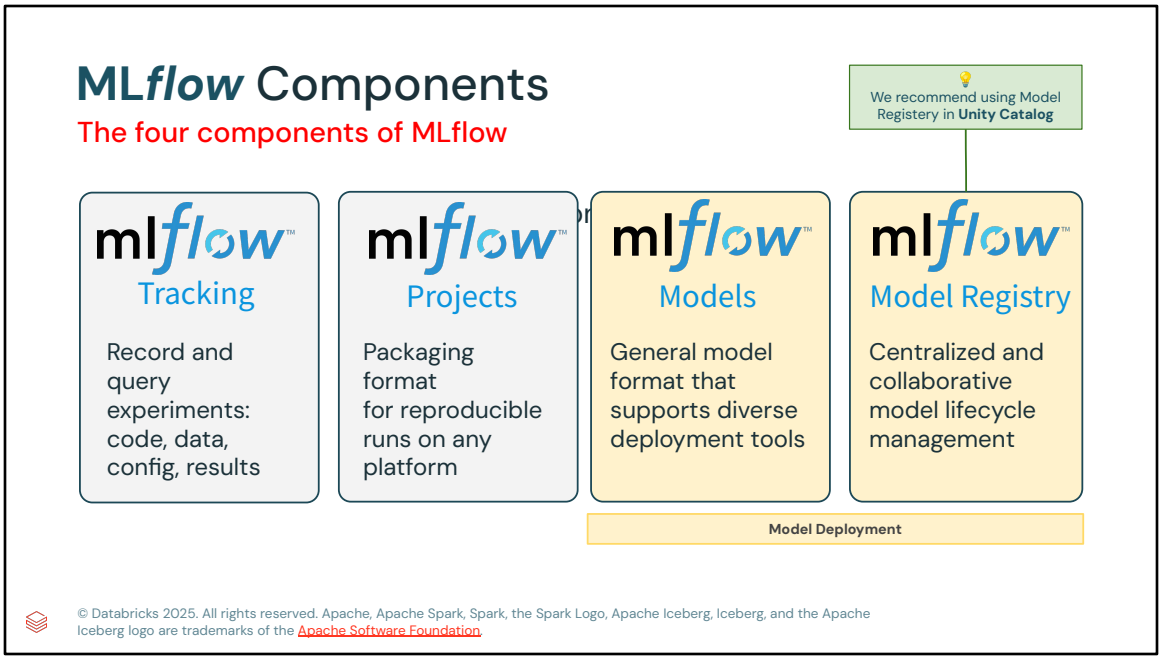
databricks

Model Deployment Fundamentals

LECTURE

Model Deployment with MLflow

This lecture covers how MLflow supports model development and deployment through its core components, model flavors, and registry. It also explains deployment modes, exporting models, and interoperability with formats like ONNX.



A quick refresher on MLflow shows that it consists of multiple components, including tracking environments and projects, which you may have already used during model training and development. For deployment, the two most important components are MLflow Models, which package trained models in a standardized format, and the Model Registry, which acts as an inventory where you can manage model versions, track their lifecycle, and apply tags.

Within the Databricks Lakehouse, the Model Registry is integrated into Unity Catalog, which we recommend using for development. The Unity Catalog model registry builds on the standard MLflow registry by adding governance, permissions, and lineage capabilities, but the underlying principles remain the same. This means that whether you are working with the Unity Catalog registry, the Workspace registry, or even the open-source MLflow registry, the concepts and workflows apply consistently.

MLflow and Model Development

What are the benefits of MLflow

Dependency & Environment Management

- Ensures that the deployment environments matches the training environment.
- Ensures that models are run consistently, regardless of where they are deployed.

Packaging Models and Code

- Any code and configuration are packaged.
- Ensures model can be deployed seamlessly without any missing components

Multiple Deployment Options

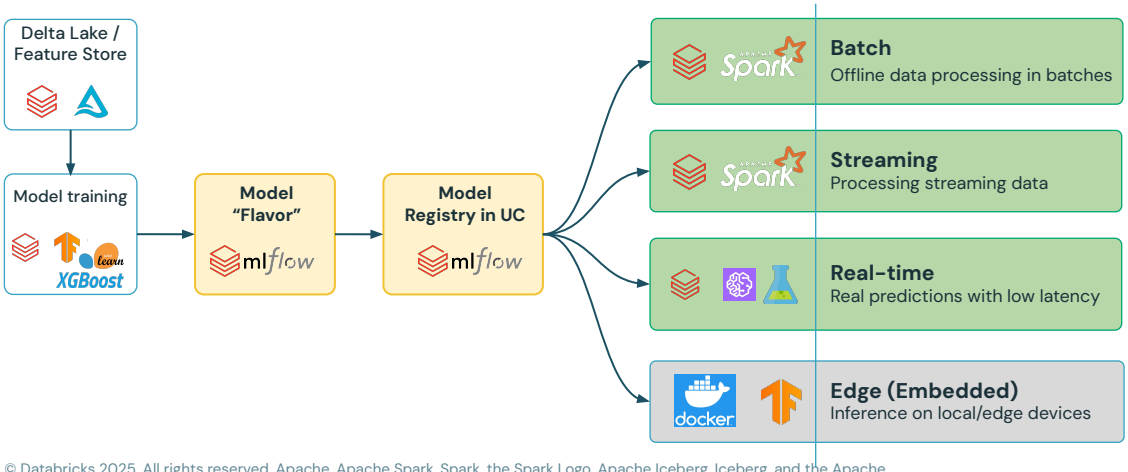
- Built-in local Flask Server with MLServer
- Remote container serving:
  - AzureML
  - AWS Sagemaker
- Kubernetes clusters
- Databricks Model Serving

You might wonder why you should create an MLflow model instead of just saving your model as a pickle file or spinning it up directly in a Flask app. The main reason is reproducibility and portability. MLflow automatically captures not only the trained model but also its dependencies and environment configurations, such as pip requirements or conda details. This ensures that when you load the model in a different machine or production environment, it runs under the same conditions as during training—without you manually managing configs.

Beyond packaging the binary model artifact, MLflow also includes any associated wrapper code, such as pre- or post-processing steps, so the full pipeline can be reused as a single artifact. Most importantly, MLflow models are interoperable across environments. Whether you deploy with MLflow's built-in serving, in a Docker container, on Kubernetes, Azure ML, or Databricks Model Serving, the same model package works without extra adjustments. This standardization makes it easier to share, reuse, and reliably deploy models across teams and platforms.

MLflow and Model Deployment Modes

Serving models for batch, streaming, real-time and edge inference



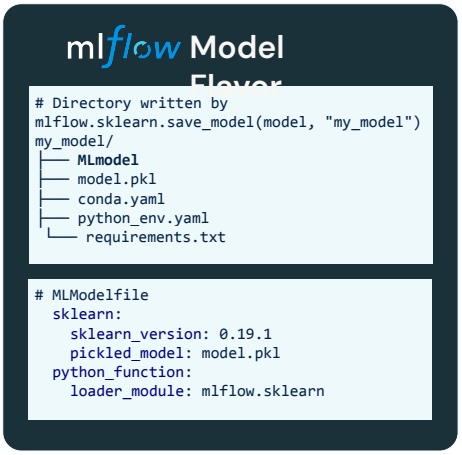
Once data is prepared and models are trained (e.g., with scikit-learn, TensorFlow, or XGBoost), they are wrapped as MLflow models using the right flavor—something Databricks automates seamlessly. When you have a winning model ready for testing or production, you push it to the **Unity Catalog Model Registry**, which manages versioning and access. From there, the registered model can easily be used for **batch inference jobs, streaming predictions** (with Structured Streaming or Lakeflow Declarative Pipelines), or **real-time serving** with Databricks Model Serving.

For **edge deployment**, models can be embedded into devices running lightweight Docker environments. To handle resource constraints, smaller models are preferred. Deep learning models (TensorFlow/Keras) can be converted to **TensorFlow Lite** for mobile/edge devices, or to the **ONNX format** for broader interoperability. While the lecture doesn't dive into full conversion code, it provides pointers for enabling these deployment scenarios.

MLflow Model ("Flavor")

A **standard format** for packaging machine learning models

- Each **MLflow Model** is a directory containing arbitrary files, together with an `MLmodel` file.
- `MLModel` file can define **multiple flavors** that the model can be viewed in.
- With MLflow Models deployment tools can understand the model.
- Model file can contain **additional metadata** such as `signature`, `input example` etc.



MLflow makes it easy to package trained models, such as scikit-learn, into a structured format. When you save a model with MLflow, it doesn't just store the serialized model file (like a pickle). It also captures metadata such as the Python version, Conda or pip dependencies, and configuration details needed to reproduce the environment. This ensures the model can be reliably loaded and executed in any environment.

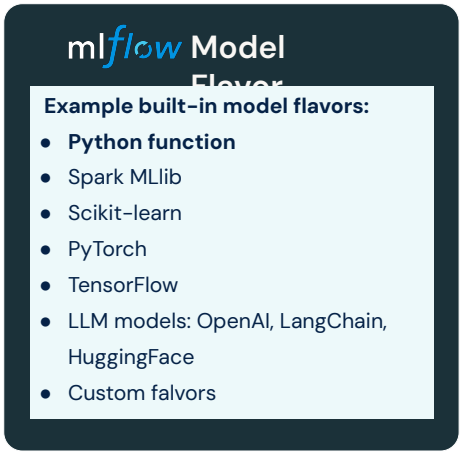
In the past, data scientists often had to manually manage pickle files, export Conda environments, and keep track of requirements, which was error-prone and hard to share. With MLflow, this entire process is automated. Instead of juggling files, you interact with the MLflow API to log, version, and retrieve models directly, making deployment seamless across Docker, cloud platforms, or Databricks Model Serving.

MLflow Model ("Flavor")

Built-in model flavors

Python Function (`mlflow.pyfunc`):

- Serves as a **default model interface** for MLflow Python models.
- Any MLflow Python model is expected to be loadable as a python function.
- Allows you to deploy models as Python functions.
- It includes all the information necessary to **load and use a model**.
- Some functions: **`log_model`, `save_model`, `load_model`, `predict`**



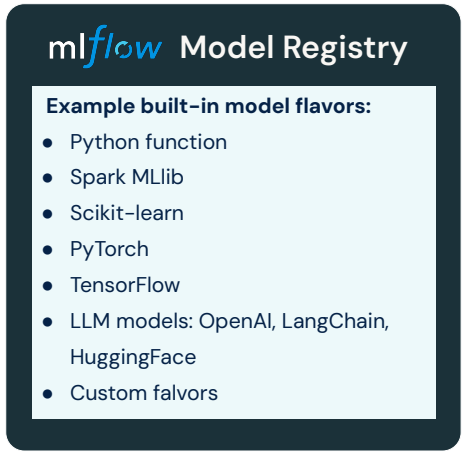
MLflow supports multiple **flavors** of models, such as Spark, scikit-learn, PyTorch, TensorFlow, and even custom ones. Each flavor defines how a model is saved, loaded, and served, making it easier to run models across different environments. Among these, the **Python function flavor** plays a central role. It acts as a default interface for MLflow models in Python, exposing methods to log models, save them as artifacts, reload them into memory, and run predictions.

The Python function flavor is also the foundation for most other MLflow flavors and enables flexibility to build **custom models**. For example, you might add preprocessing before prediction or post processing after prediction. By creating a custom class that extends the Python function flavor, you can wrap extra logic around your model while still benefiting from MLflow's standardized tracking and deployment.

MLflow Model Registry

A centralized model store

- Deploy and organize models.
- Model **versioning.**
- Model lifecycle management with **aliases**. Example: **`champion`** for the **`Production`** stage.
- Collaboration and permission management.
- Full model lineage.
- Tagging and annotations.



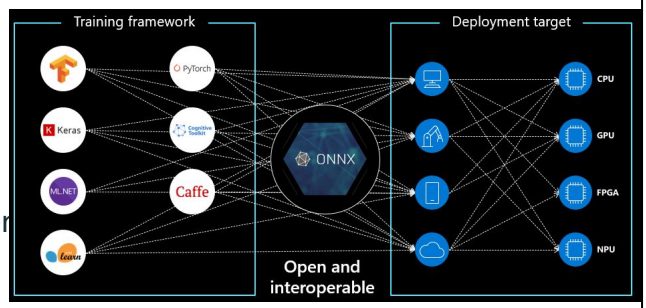
Once a model is trained and logged as an MLflow flavor, it carries all the necessary metadata and artifacts for reuse. At this point, managing versions, applying tags, and setting up conventions like a champion–challenger approach becomes important. This allows you to track how the model evolves over time and ensure that the right version is being used in production.

The **model registry** streamlines this process by providing a central place to store, organize, and control access to models. Instead of manually tracking models through spreadsheets or ad hoc systems, the registry maintains lineage, version history, and permissions automatically. With Unity Catalog's model registry, these capabilities extend further by embedding governance and access controls, making it easy to share models across teams while ensuring consistency and security.

Exporting Models in MLflow

Deployment environments

- In some cases, such as on the “Edge”, models need to be deployed in specific hardwares.
- Environment characteristics:
  - Offline / unreliable connectivity
  - Latency requirements
  - Security and compliance
- Export the MLflow model for inferer
  Lite.



When it comes to **exporting models**, the target environment matters—whether it’s the cloud, on-prem servers, mobile devices, or edge hardware. Most MLflow flavors run smoothly on cloud or workstation setups, but lightweight or specialized devices often require conversion. For example, models can be exported to the **ONNX format**, an open standard for model exchange that enables deployment across different hardware types, including CPUs, GPUs, and specialized accelerators.

Similarly, deep learning models built in TensorFlow or Keras can be converted to **TensorFlow Lite**, a format optimized for mobile and edge devices. These conversions ensure that models run efficiently under hardware constraints. That said, in the majority of cases—roughly 80%—models are deployed in cloud or on-prem environments, where direct use of MLflow flavors is sufficient without extra conversion.

MLflow and ONNX

A Keras conversion and logging example

- Convert model to ONNX format.
- Save ONNX model as ONNX flavor.
- Scoring: use ONNX Runtime or convert to native flavor.

Example

```
# Convert and log model as ONNX

import onnxmltools
onnx_model =
onnxmltools.convert_keras(model)
mlflow.onnx.log_model(onnx_model,
"onnx-model")
```

```
# Read and score model
import onnxruntime
session =
onnxruntime.InferenceSession(model.Serialize
ToString())
input_name = session.get_inputs()[0].name
predictions = session.run(None, {input_name:
data_np.astype(np.float32)})[0]
```

As an example, models can be easily converted and logged in the **ONNX format** using the `onnx` or `onnxmltools` libraries. For instance, you can take a Keras model, convert it, and then log it directly as an ONNX artifact in MLflow. This makes the process straightforward while ensuring compatibility across environments.

Thank you for completing this lesson and continuing your journey to develop your skills with us.