Microsoft

Machine Learning Operations (MLOps)

CS5304 – Big Data Services & MLOps

Learning Units covered in this Module

- Introduction to MLOps
- Building Blocks for MLOps
- MLOps Maturity

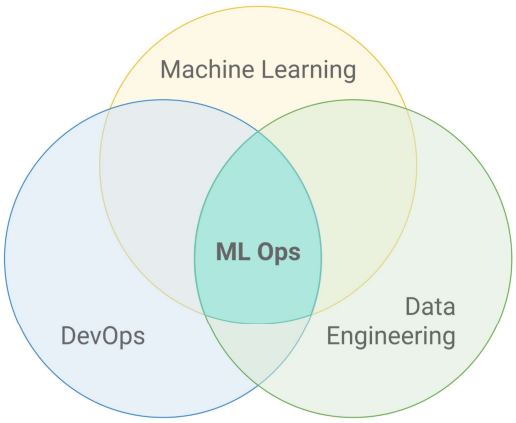
Objectives

After completing this learning, you will be able to:

- Describe MLOps and its benefits to business
- Describe the goal of MLOps and its components
- Describe MLOps in Practice
- Describe MLOps Maturity Levels

MLOps

"ML Ops: Machine Learning as an Engineering Discipline"



"MLOps or ML Ops is a set of practices that aims to deploy and maintain machine learning models in production reliably and efficiently"

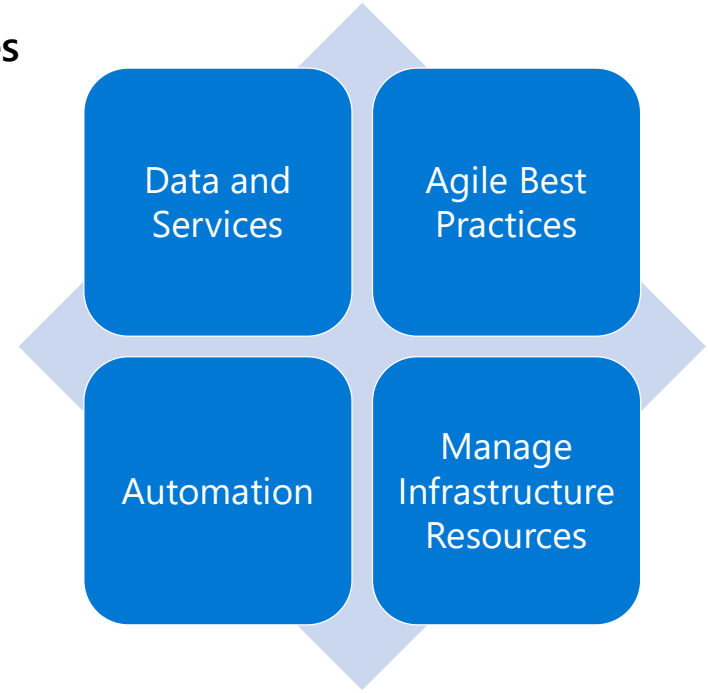
MLOps is practiced between Data Scientists, DevOps, and Machine Learning engineers to transfer the trained model to production systems to serve. Similar to DevOps or DataOps approaches, MLOps seeks to increase automation and improve the quality of production models, while also focusing on business and regulatory requirements. While MLOps started as a set of best practices, it is slowly evolving into an independent approach to ML lifecycle management. MLOps applies to the entire lifecycle - from integrating with model, orchestration, and deployment, for monitoring, diagnostics, governance, and business metrics.

According to Gartner, MLOps is a subset of ModelOps. MLOps is focused on the operationalization of machine learning models, while ModelOps covers the operationalization of all types of AI models.

MLOps Benefits

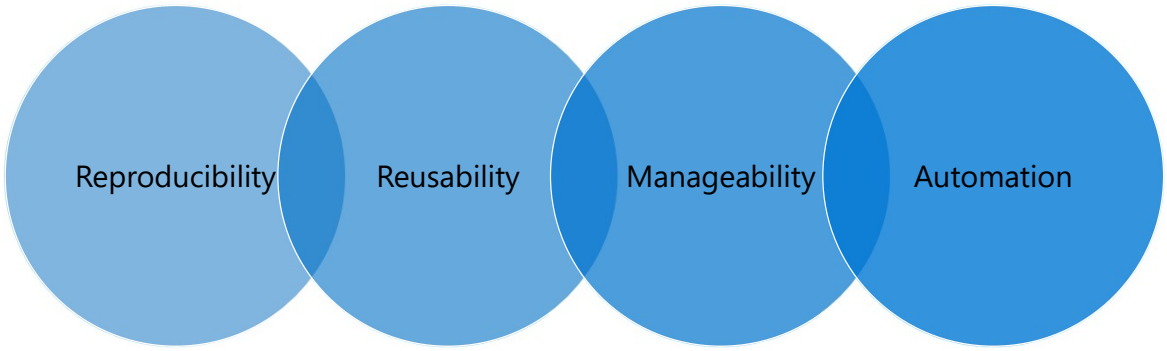
- Faster time to market of ML-based solutions
- More rapid rate of experimentation, driving innovation
- Assurance of quality, trustworthiness and ethical AI

MLOps Core Principles



- Connect Data and Services – DevOps success depends on how well platforms of data and existing/new services can be integrated, adapting to changing circumstances.
- Automation – Automation needs to be considered in the context of the above, to ensure constant, consistent, and efficient delivery of business value.
- Manage Infrastructure Resources – Applications will be deployed to an increasingly commoditized, flexible, target environment of infrastructure and platform-level services.

MLOps Goal



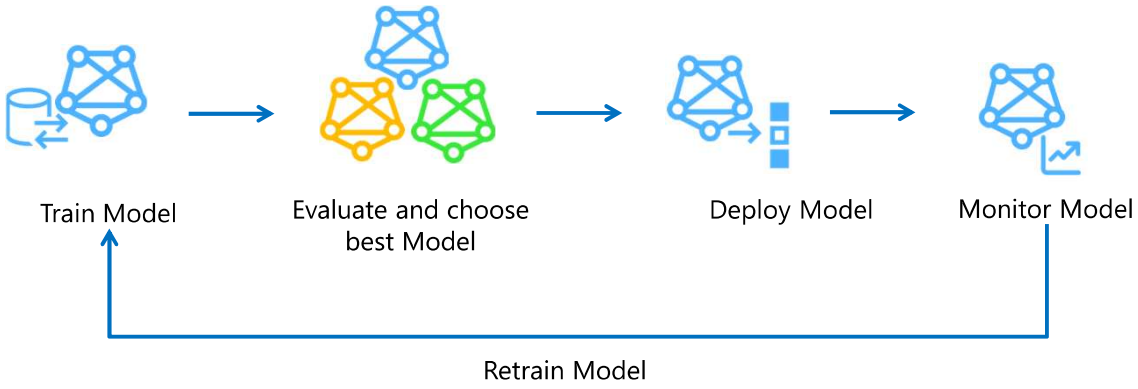
Reproducibility – as with software configuration management and continuous integration, ML pipelines and steps, together with their data sources, code, models, libraries, and SDKs, need to be versioned and maintained such that they can be reproduced exactly as previously.

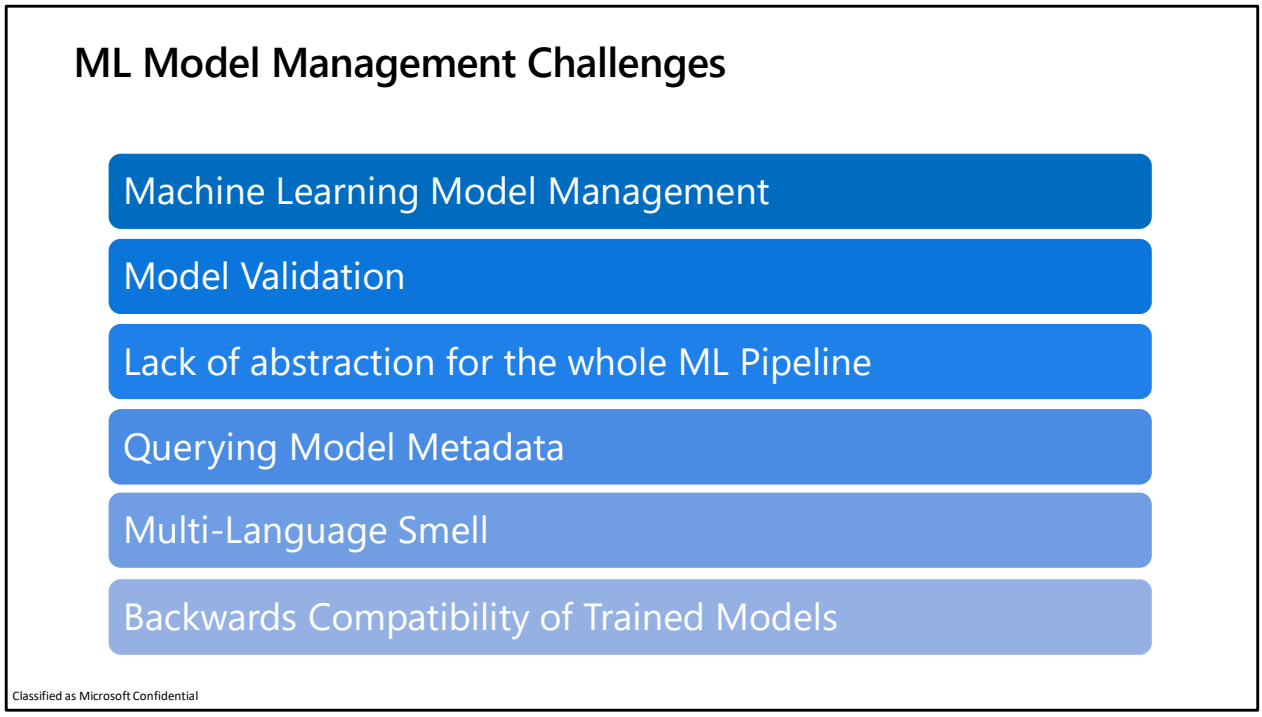
Reusability – to fit with principles of continuous delivery, the pipeline needs to be able to package and deliver models and code consistently into training and target environments, such that the same configuration can be repeated with the same results.

Manageability – the ability to apply governance, tracking changes to models and code throughout the development lifecycle, project tracking (for example through sprints), and enabling managers to measure and oversee both progress and valuedelivery.

Automation – as with DevOps, continuous integration and delivery require automation to assure rapid and repeatable pipelines, particularly when these are augmented by governance and testing (which can otherwise create a bottleneck).

MLOps in Practice





Here we highlight some of model management challenges.

First challenge is Machine Learning Model Management, which considers how we manage model parameters / weights.

Second, there is mode validation, the ability to able back-test the performance of models over time. Models evolve over time as data changes, hardware improvement or algorithm changes. Every time such a change happens, model needs to be retrained and model performance must be re-validated. These validations introduce a number of challenges for example using the same code and same data across different models to be able to reproduce the same result.

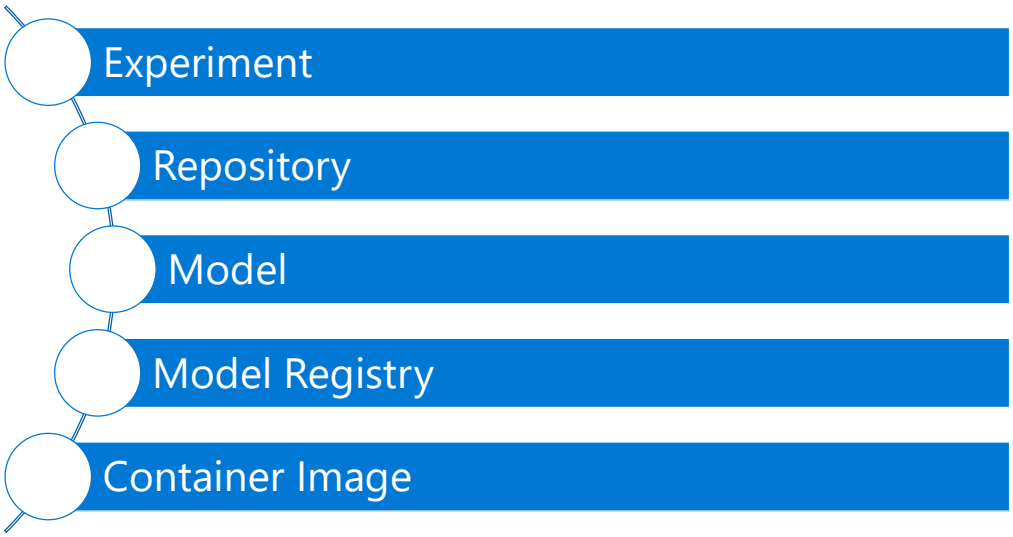
In real-world, there are cases where different systems need to be "glued" together in ML deployments, and often require external orchestration frameworks that coordinate the workloads. This situation has big negative impact for model management tasks: It complicates the extraction of metadata corresponding to a particular pipeline it makes reproduciability and automation of model selection difficult as many different systems have to be orchestrated, and as a consequence makes it hard to automate model validation.

In order to automate and accelerate model lifecycle management, it is important to query metadata like hyperparameters, evaluation scores as well as the datasets on which they were trained and validated). This metadata is required during models selection in order to decide which one to put into production, or for getting a review on training progress, or for identifying bottleneck so we can decide where we want to spend more time finetuning our model. Additionally, a centralized metadata store builds a fundation for a more efficient process in enterprise solution and can also be used to automate some steps or processes.

Another hard challenge in model management is called "Multiple-Language Smell", it comes from the fact that end-to-end ML applications often written in different programming languages with different frameworks. Many popular libraries are written in python with different frameworks. It's hard to keep heterogeneous code consistent as automatic tools tools can only inspect either the python or the JVM part of the code, and will not be able go across the language barrier. To make things even worse, this makes it even harder to deploy at later time, as they require setups with many different components that need to be orchestrated. For example., a Spark cluster must be started and running for pre-processing or feature engineering, afterwards the data might need be moved to a another virtual machine for model training, and the cluster must be shut down afterwards, we will then need another virtual machine to run inference. This raises the challenge of how to efficient and reliable exchange of code, data and model between the different components of the system written in different languages and different frameworks.

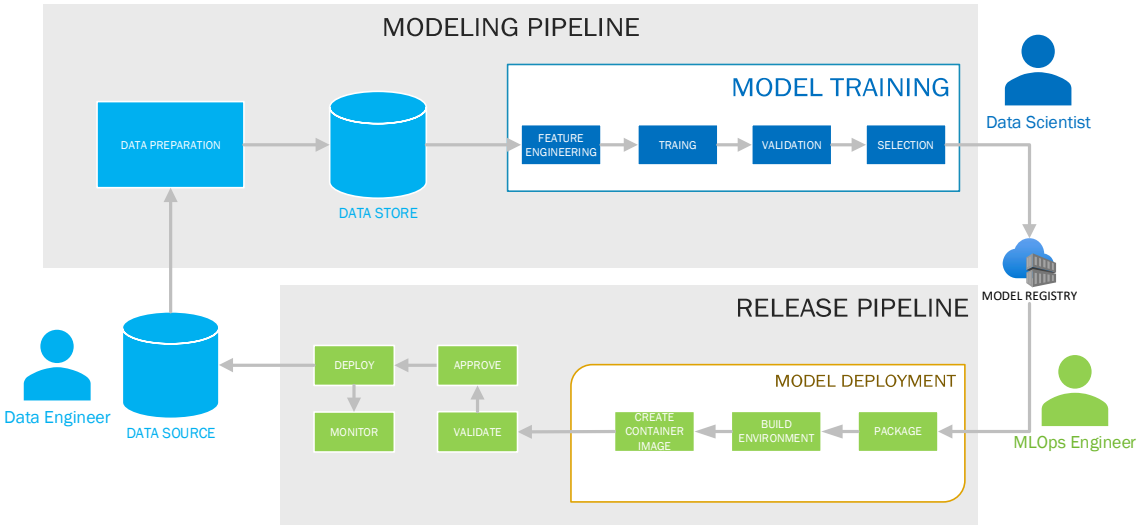
All trained models require backwards compatibility. For example, a model that was trained last month or last quarter should still be working today, in particular when trained models might be used in production deployments.In the meantime, we also want to reproduce the exact same result, or a similar result can be achieved, or in the worst case that the model can still run. Ensuring these conditions comes with several challenges on model management.For example, when a model is deployed and served in production, making sure that the exact same result sometime can be retrieved is a strict requirement. According to this, storing all the components that trained a model is fundamental piece in being able to guarantee backwards compatibility and long-term usage. We need to store the data , data transformations, the code implementation, the inference code of calling the model, its configuration and its dependencies in details

MLOps Terms



- Experiment – An activity sequence that enables a hypothesis to be tested and validated iteratively. Outputs of a given iteration need to be stored so they can be assessed, compared, and monitored for audit purposes.
- Repository – A common, version-controlled storage resource (e.g. Git) for data, model, and configuration schemas, managing dependencies between models, libraries, and other resources.
- Model Registry – A logical picture of all elements required to support a given ML model, across its development and operational pipeline.
- Model – Packaged output of an experiment that can be used to predict values or built on top of (via transfer learning).

MLOps and Roles

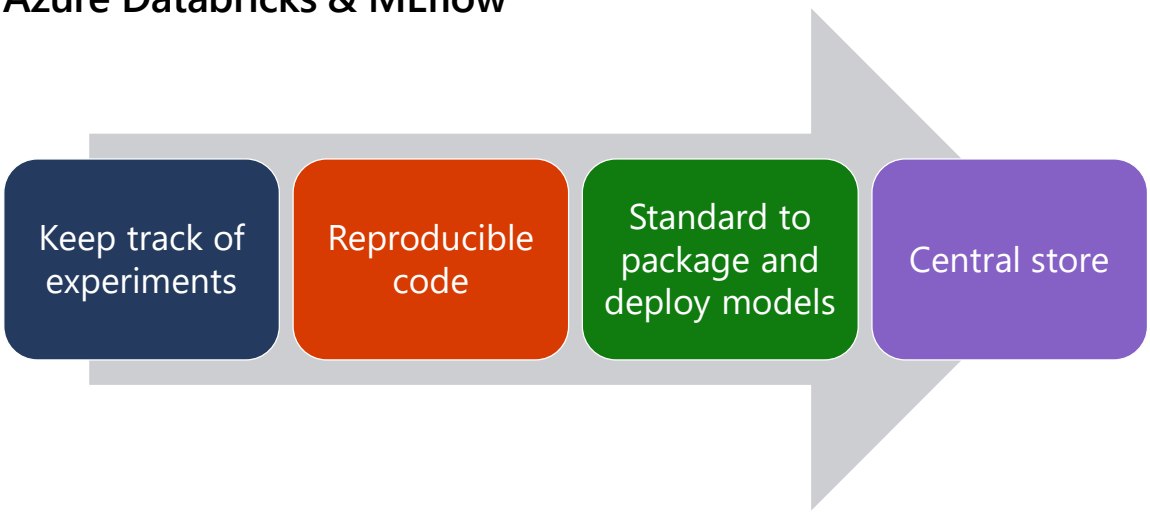


MLOps Maturity

Level	Automated Training	Automated Deployment	Central Repository for Model Training and Model Registry
No MLOps			
Automated Training	X		X
Automated Deployment		X	X
Full Automation	X	X	X
	Machine Learning Pipeline	ML Model Registry / Container Registry	Azure ML Workspace

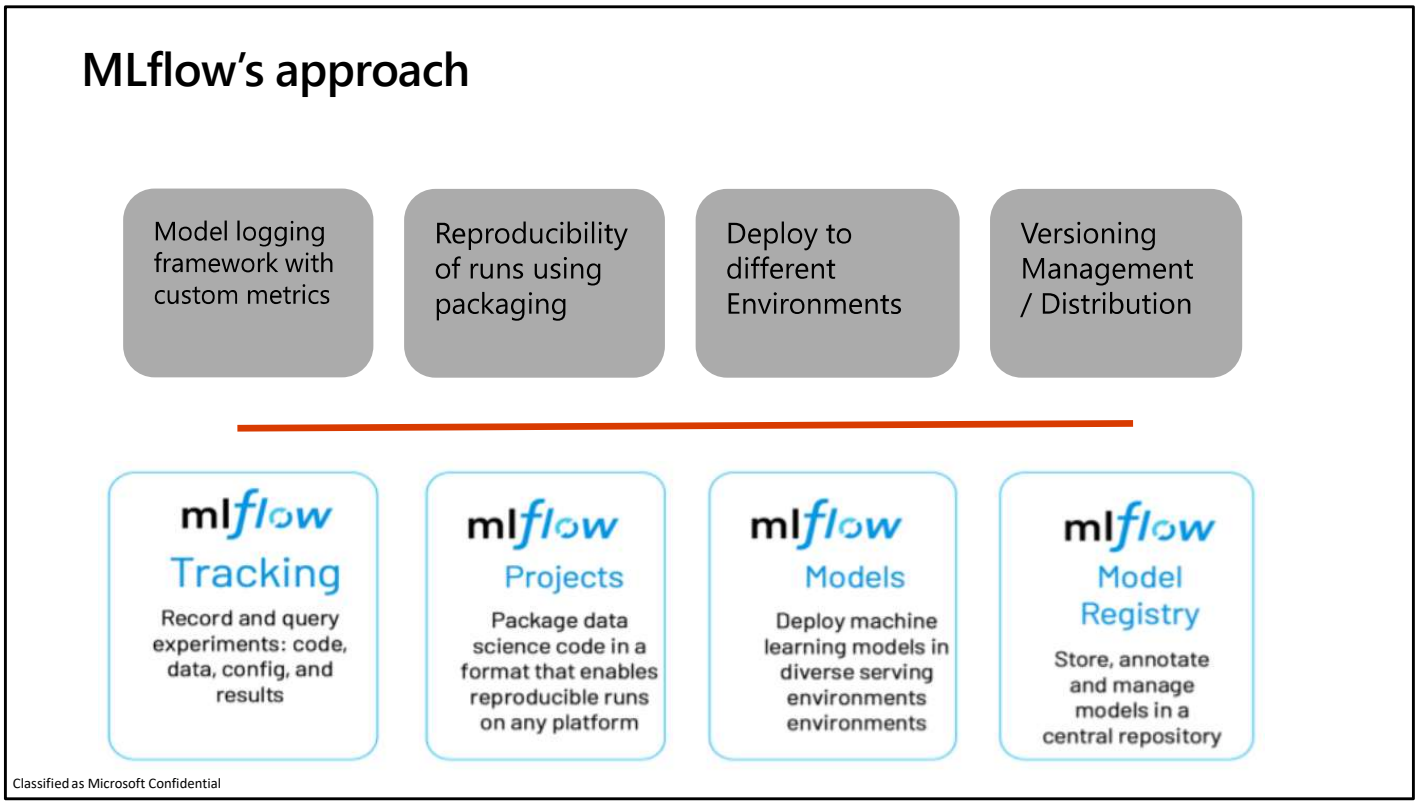
The purpose of this maturity model is to help clarify the Machine Learning Operations (MLOps) principles and practices. The maturity model shows the continuous improvement in the creation and operation of a production level machine learning application environment. You can use it as a metric for establishing the progressive requirements needed to measure the maturity of a machine learning production environment and its associated processes.

Azure Databricks & MLflow



MLflow is an open source platform to manage the ML lifecycle, including experiments, reproducibility, deployment, and a central model registry targeting some of the challenges I mentioned earlier. It is designed to work with any machine learning libraries and only require minimal changes to integrate into an existing codebase. At the same time, MLflow aims to take any codebase written in its format and make it reproducible and reusable using its standard format.

During data science process, we might need to consider using individual ML libraries so we can provide the best solutions to the problems, it could become a challenge like we discussed in previous slide. MLflow is designed to lets data scientist train, reuse, and deploy models with any library and package them into reproducible processes that other data scientists can use as a “black box,” without even having to know the details of the libraries being used.



MLflow currently offers four components

MLflow Tracking is an API and UI for logging parameters, code versions, metrics, and artifacts when running your machine learning code and can also be used later for reports and visualizations. You can use MLflow Tracking in any environment (for example, a standalone script or a notebook) to log results to local files or to a server, then compare multiple runs. Data science team can also use it to compare results from different users.

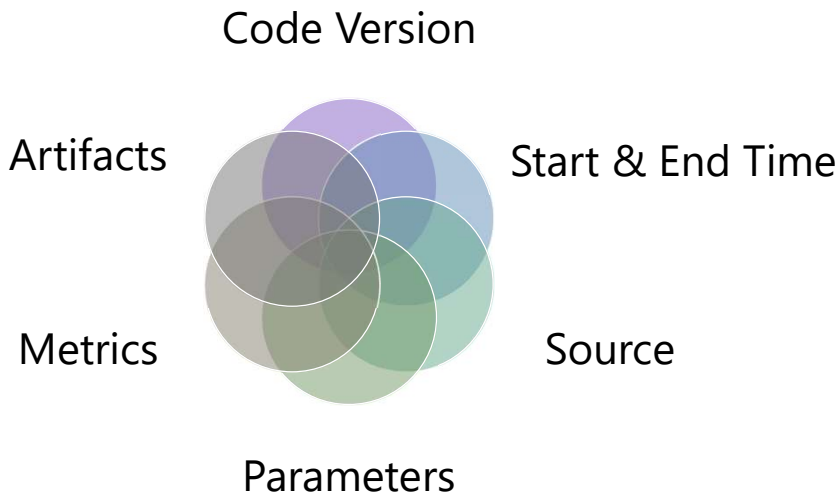
MLflow Projects are a standard format for packaging reusable data science code. Each project is simply a directory with code or a Git repository, and uses a descriptor file or simply convention to specify its dependencies and how to run the code. For example, projects can contain a conda.yaml file for specifying a Python Conda environment. When you use the MLflow Tracking API in a Project, MLflow automatically remembers the project version (for example, Git commit) and any parameters. You can easily run existing MLflow Projects from GitHub or your own Git repository, and chain them into multi-step workflows.

MLflow Models offer a convention for packaging machine learning models in multiple flavors, and a variety of tools to help you deploy them. Each Model is saved as a directory containing arbitrary files and a descriptor file that lists several "flavors" the model can be used in. For example, a TensorFlow model can be loaded as a TensorFlow DAG, or as a Python function to apply to input data. MLflow provides tools to deploy many common model types to different platforms: for example, any model supporting the "Python function" flavor can be deployed to a Docker-based REST server, to cloud platforms such as Azure ML service, and as a user-defined function in Apache Spark for batch and streaming inference. If you output MLflow Models using the Tracking API, MLflow also automatically remembers which Project and run they came from.

MLflow Registry offers a centralized model store, set of APIs, and UI, to manage the full lifecycle of an MLflow Model. It provides model lineage (which MLflow experiment and run produced the model), model versioning, stage transitions (for example from staging to production or archiving), and annotations.

These components can be used individually, or can be used together.

MLflow Tracking

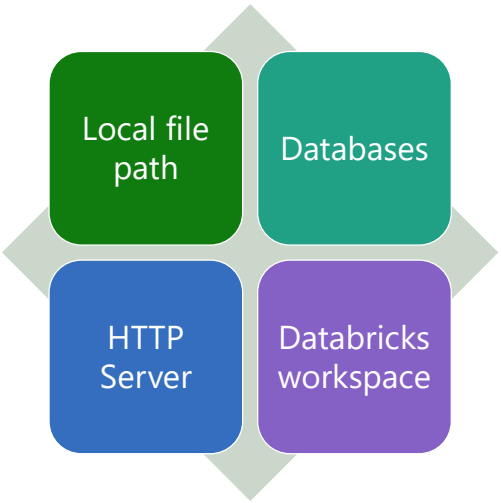


•An MLflow *experiment* is the primary unit of organization and access control for MLflow runs; all MLflow runs belong to an experiment. Experiments let you visualize, search for, and compare runs, as well as download run artifacts and metadata for analysis.

•An MLflow *run* corresponds to a single execution of model code. Each run records the following information:

- **Source**: Name of the notebook that launched the run or the project name and entry point for the run.
- **Version**: Notebook revision if run from a notebook or Git commit hash if run from an MLflow project
- **Start & end time**: Start and end time of the run.
- **Parameters**: Model parameters in a key-value format
- **Metrics are** Model evaluation metrics saved also in key-value pairs.
- **Tags**: are run metadata
- **Artifacts**: are Output files in any format. For example, training curves plot

Potential Tracking Location



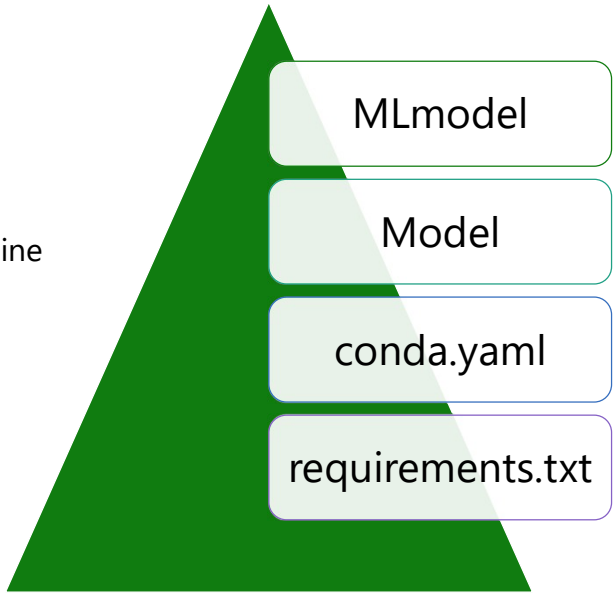
The Tracking API communicates with an Mlflow tracking server which can be one of the 4 choices.

- Local file path, where data is just directly stored on loca file system.
- MLflow supports the databases like sql server, mysql, sqlite, and postgresql
- HTTP server hosting an Mlflow tracking server.
- Databricks workspace

MLflow also supports distributed architectures, where the tracking server, backend store, and artifact store reside on remote hosts.

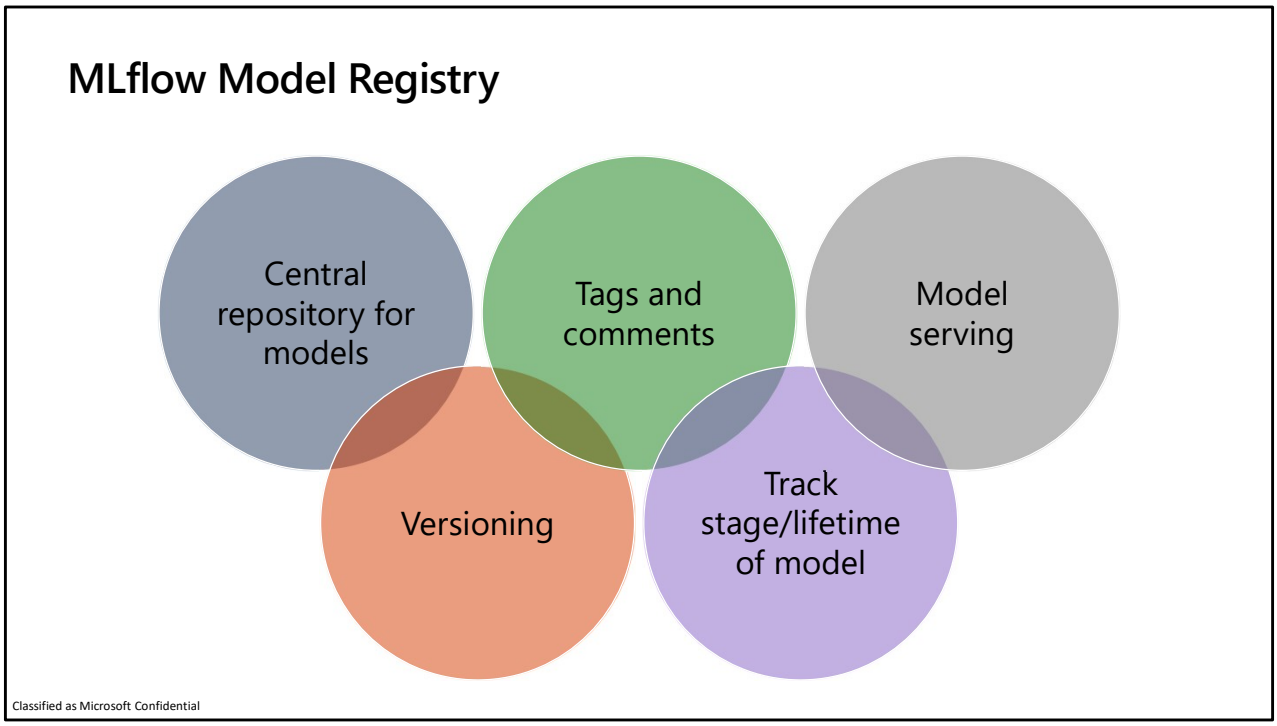
MLflow Models

A standard format for packaging machine learning models.



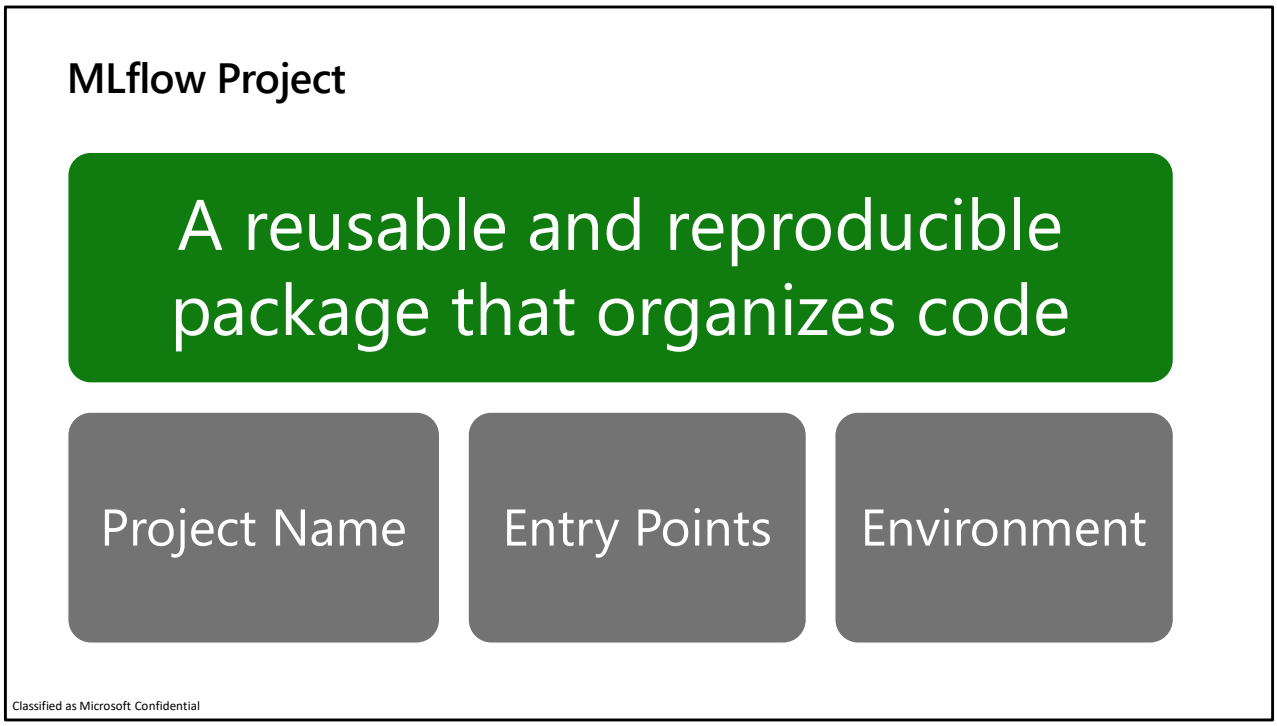
Each MLflow Model is a directory containing arbitrary files, together with an MLmodel file in the root of the directory that can define multiple *flavors*.

Flavors are the key concept that makes MLflow Models powerful: they are a convention that deployment tools can use to understand the model, which makes it possible to write tools that work with models from any ML library without having to integrate each tool with each library. MLflow defines several "standard" flavors that all of its built-in deployment tools support, such as a "Python function" flavor that describes how to run the model as a Python function.



The Model Registry introduces a few concepts that describe and facilitate the full lifecycle of an MLflow Model.

An MLflow Model is created from an experiment or run that is logged with one of the model flavor's log_model methods. Once logged, this model can then be registered with the Model Registry. Each registered model has a unique name, contains versions, associated transitional stages, model lineage, and other metadata.Each registered model can have one or many versions. When a new model is added to the Model Registry, it is added as version 1. Each new model registered to the same model name increase the version number. Each distinct model version can be assigned one stage at any given time. MLflow provides predefined stages for common use-cases such as *Staging*, *Production* or *Archived*. You can transition a model version from one stage to another stage. You can annotate the top-level model and each version individually using Markdown, including description and any relevant information useful for the team such as algorithm descriptions, dataset used or methodology.



At the core, MLflow Projects are packages for organizing and describing your code to let other data scientists (or automated tools) run it. Each project is simply a directory of files, or a Git repository, containing your code.

Name for the project.

Entry Points are commands that can be run within the project, and information about their parameters. Most projects contain at least one entry point that you want other users to use. Some projects can also contain more than one entry point: for example, you might have a single Git repository containing multiple featurization algorithms. You can also call any .py or .sh file in the project as an entry point. If you list your entry points in a MLproject file, however, you can also specify parameters for them, including data types and default values.

Environment is software environment that should be used to execute project entry points. This includes all library dependencies required by the project code.