# Components

MLflow has 4 main components:

1. **Tracking** – log experiments and metrics
2. **Projects** – packaging code
3. **Models** – managing ML models
4. **Registry** – central store for versioned models

## MLflow Tracking

Tracking is a centralized experiment logger where you track:

* **Parameters:** Hyperparameters like learning rate, batch size, etc.
* **Metrics:** Accuracy, loss, AUC, any numeric measure
* **Artifacts:** Model files, plots, datasets, any output file
* **Source info:** Git commit hashes, code version, who ran it, when

You log these from your training scripts with the MLflow client API (mlflow.log\_param(), mlflow.log\_metric(), etc.).

You can then visualize and compare experiments in the MLflow UI.

For tracking we are using 3 components:

* A **backend store**
* An **artifact store**
* MLflow Server (**Tracking Server**)

### A backend store

* usually a SQL DB like MySQL, Postgres, or SQLite
* Stores metadata:
  + Experiment info
  + Run UUIDs
  + Parameters
  + Metrics
  + Tags
  + Run start/stop times

### An artifact store

* For example Azure Blob Storage, AWS S3, local FS
* Stores files:
  + Saved models
  + Plots
  + Logs

### MLflow Server (Tracking Server)

* Provides UI
* Clients (your scripts or projects) send data here
* It orchestrates:
  + Logging to the **backend store**
  + Uploading to the **artifact store**
  + Serving the **web UI** to browse runs

We set up a Tracking Server with a single command:

A screenshot of a computer program

AI-generated content may be incorrect.

## MLflow Project

This helps you **package your ML code** with all dependencies to make it reproducible and shareable.

That can include code for:

* Training models
* Preprocess data
* Run evaluations

We can specify parameters when running a project which will be used by the code (like hyperparameters).

### Entry points

An entry point consist of a command and parameters used in that command. That command executes a function related to our ML model, for example a function for:

* Training models
* Preprocess data
* Run evaluations

Entry points are defined in a MLproject file. An example of how it looks like is in the next section ‘MLproject file’.

### MLproject file

The MLproject file is a YAML file which describes:

* Project name
* Environment (conda, docker, or system)
* Entry points (commands for running scripts)
* Parameters (if any)

It can look for example like that:

A screen shot of a computer program

AI-generated content may be incorrect.

### MLflow project content

Here is a typical folder structure for a MLflow project:

A screenshot of a computer program

AI-generated content may be incorrect.

### Docker

We can use Docker for preparing an environment where scripts (entry points) from the MLflow project will be executed.

In the MLproject file we can specify either a Dockerfile or a Docker image which will be used to prepare that environment.

Docker image which we want to use in the MLflow project can be saved either on our local computer or in a remote registry.

### Deploying on Kubernetes

In order to run our MLflow project on Kubernetes we need to include a backend\_config.yaml file in the MLflow project repository (in the same folder as Dockerfile) and provide the following parameters:

A screenshot of a computer

AI-generated content may be incorrect.

* kube-context
  + Which kube context (which Kubernetes cluster) we want to use.
  + It is required only if we are working with multiple clusters.
* Service\_account
  + Name of the Service Account to use.
  + It is required only if we are pulling a Docker image from a private repository and we need to authenticate.
  + That Service Account needs to have assigned a proper role allowing for reading Kubernetes Secrets
  + It will be used to read data from a Secret which is used to authenticate to a private container repository.
* Env.MLFLOW\_TRACKING\_URI
  + This environment variable is needed if we are not use the default local Tracking Server but instead we have it deployed somewhere else (Kubernetes, VM, Databricks)

Then we run our project using this command:

* mlflow run . -b kubernetes -P lr=0.01 -P epochs=5 --backend-config backend\_config.yaml

## MLflow Models

This component standardizes how you **package, save, and load ML models** regardless of framework (TensorFlow, PyTorch, Scikit-learn, XGBoost, etc.).

* Models are saved in a standard format, with metadata describing how to load and serve them
* Supports multiple deployment flavors (options for loading and serving a model) like Python function, REST API, Spark UDF, or even batch transforms

Saved models can be stored anywhere. One option is MLflow Registry.

## MLflow Registry

A **centralized model store** where you manage the lifecycle of your ML models:

* **Versioning:** Keep track of every model version registered
* **Stage transitions:** Move models through stages like Staging, Production, Archived
* **Annotations & comments:** Add notes or tags on model versions
* **Deployment integration:** Some frameworks can deploy directly from the registry

It basically brings **model governance and collaboration** to your ML pipeline.

# Experiments and runs

Experiments are a way to organize models. Each experiment is a group of models.

Every time we create a model:

* It gets assigned to an experiment
* It gets assigned a run ID

Run IDs are unique across all experiments. This is a way to identify models.

## Saving models

When creating and saving a model we can assign it to a specific experiment and run:

A screen shot of a computer program

AI-generated content may be incorrect.

Model’s name specified as the artifact\_path argument will be a folder name where model will be stored.

We can use that name when loading a model.

In artifact store we have this file structure:

A computer screen shot of a program

AI-generated content may be incorrect.

## Loading models

When we load a model, we can identify it using run ID and model name:

* mlflow.sklearn.load\_model(f"runs:/{run\_id}/model\_name")

In order to get run ID we can query all the runs as described in the next ‘Querying runs’ section in this document.

## Querying runs

We can use Python to get information about different runs. Below is example of how to do this based on an experiment name:

A screen shot of a computer program

AI-generated content may be incorrect.

In the search\_runs function we have different options for querying runs data, for example we can sort results on a start time of a run in a descending order:

A screen shot of a computer code

AI-generated content may be incorrect.

The ‘runs’ variable is a DataFrame with information about runs. It contains such a columns as:

* Run ID
* Experiment ID
* Status
* Start and End Time
* Tags
* Params
* Metrics

We can use this data to find the run ID which we are looking for.