MLflow
Complete Guide

Experiment Tracking & Model Registry

A Comprehensive Guide to MLflow for Machine Learning
Experiment Tracking, Model Registry, and Production Deployment

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Introduction to MLflow

1.1 What is MLflow?

MLflow is an open-source platform designed to manage the complete machine learning lifecycle. It provides tools for experiment tracking, model versioning, deployment, and collaboration.

Core MLflow Components

1. **MLflow Tracking**: Log and query experiments (parameters, metrics, artifacts)
2. **MLflow Projects**: Package ML code in reusable, reproducible format
3. **MLflow Models**: Manage and deploy models from various ML libraries
4. **MLflow Model Registry**: Centralized model store for managing model lifecycle

1.2 The Experimentation Challenge

In machine learning projects, multiple experiments are conducted at each pipeline stage:

- **Pre-Processing**: E1, E2, E3, ...
  - E1: Handling outliers using IQR
  - E2: Handling outliers using Isolation Forest
  - E3: Different scaling techniques
- **Feature Engineering**: E1, E2, E3, ...
  - Different feature selection methods
  - Various feature transformation techniques
  - Multiple feature combinations
- **Model Selection**: E1, E2, E3, ...
  - Random Forest vs XGBoost vs Neural Networks
  - Different algorithm families
- **Hyperparameter Tuning**: E1, E2, E3, ...
  - Grid search combinations
  - Random search trials
  - Bayesian optimization iterations

Important Note

After conducting numerous experiments across all stages, we need to identify the combination that provides the best results. Industry-grade tools like **DVC** and **MLflow** are essential for managing this complexity.

2 DVC vs MLflow Comparison

2.1 Evolution and Purpose

2.1.1 DVC (Data Version Control)

- **Initial Purpose**: Data versioning only
- **Evolution**: Added experiment tracking after gaining popularity
- **Inspiration**: Experiment tracking inspired by MLflow
- **Maturity**: Newer to experiment tracking compared to MLflow

2.1.2 MLflow

- **Purpose-Built**: Designed from the ground up for ML lifecycle management
- **Maturity**: More mature and feature-complete for experiment tracking
- **Flexibility**: Can be used independently without Git

2.2 Key Differences

DVC	MLflow
Must be used with Git	Can be used without Git
UI is basic and less intuitive	Rich, user-friendly web UI
Experiment history is local only	Allows team-wide collaboration
Primarily for data versioning	Comprehensive ML lifecycle management
No centralized experiment tracking	Centralized tracking server available
Limited visualization capabilities	Advanced visualization and comparison tools

2.3 Why MLflow is Preferred

Due to the following advantages, **MLflow is given higher priority** in industry for experiment tracking:

1. **Superior UI/UX**: Intuitive web interface for experiment comparison
2. **Team Collaboration**: Centralized server allows entire team to view and compare experiments
3. **Independence**: Works standalone without requiring Git
4. **Maturity**: More stable and feature-complete for experiment tracking
5. **Comprehensive Features**: Covers entire ML lifecycle, not just data versioning

2.4 MLflow Capabilities Beyond Experiment Tracking

- **Visualization**: Interactive plots and metrics comparison
- **Generative AI**: Support for LLM tracking and evaluation
- **Evaluation**: Built-in model evaluation frameworks
- **Model Registry**: Complete model lifecycle management

- **Serving**: Model deployment and serving capabilities
- **Models**: Standardized model format for various frameworks

3 Getting Started with MLflow

3.1 Installation and Setup

3.1.1 Step 1: Create Repository

```
# Create a new repository on GitHub
# Clone it locally
git clone https://github.com/username/mlflow-project.git
cd mlflow-project
```

3.1.2 Step 2: Install MLflow

```
# Install MLflow
pip install mlflow

# Verify installation
mlflow --version
```

3.1.3 Step 3: Launch MLflow UI

```
# Start MLflow UI
mlflow ui
```

What happens:

- MLflow UI starts at `http://127.0.0.1:5000`
- Creates `mlflow.db` file (SQLite database for metadata)
- Ready to receive experiment data

3.1.4 Step 4: Create Project Structure

```
# Create source folder
mkdir src
cd src
```

Project structure:

```
mlflow-project/
+-- src/
|   +-- main_1.py
|   +-- main_2.py
|   +-- ...
+-- mlflow.db
+-- mlartifacts/
+-- mlruns/
```

3.2 Understanding MLflow Storage

Storage Locations

Where artifacts are stored depends on setup:

Setup	Artifact Folder
File-based tracking	mlruns/
MLflow server	mlartifacts/
Dagshub (remote)	Remote (no local folder)

Key Point: Artifacts are stored differently based on the artifact store configuration. Local tracking uses `mlruns/`, while server-based tracking uses `mlartifacts/`.

4 Basic MLflow Experiment Tracking

4.1 First MLflow Script

main_1.py - Basic Experiment

```python
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

mlflow.set_tracking_uri("http://127.0.0.1:5000")

# Load Wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# Train test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42
)

# Define the params for RF model
max_depth = 10
n_estimators = 5

with mlflow.start_run():
    rf = RandomForestClassifier(
        max_depth=max_depth,
        n_estimators=n_estimators,
        random_state=42
    )
    rf.fit(X_train, y_train)

    y_pred = rf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

    mlflow.log_metric('accuracy', accuracy)
    mlflow.log_param('max_depth', max_depth)
    mlflow.log_param('n_estimators', n_estimators)

    # Creating a confusion matrix plot
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(6,6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=wine.target_names,
                yticklabels=wine.target_names)
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title('Confusion Matrix')

    # save plot
```

```
    plt.savefig("Confusion-matrix.png")

    # log artifacts using mlflow
    mlflow.log_artifact("Confusion-matrix.png")
    mlflow.log_artifact(__file__) # for saving the current
   script

    # tags
    mlflow.set_tags({
        "Author": 'Sujil S',
        "Project": "Wine Classification"
    })

    # Log the model
    mlflow.sklearn.log_model(rf, "Random-Forest-Model")

    print(accuracy)
```

4.2 Running the Script

```
# Run the script
python src/main_1.py
```

What happens:

1. Creates `mlartifacts/` folder
2. Stores plot, model, and Python script
3. Run is stored in default experiment (ID: 0)
4. Unique Run ID is generated
5. All data visible in MLflow UI at `http://127.0.0.1:5000`

5 What Can Be Logged in MLflow

5.1 Complete Logging Capabilities

5.1.1 1. Metrics

Numeric values tracked over time:

- **Accuracy**: Model accuracy across runs
- **Loss**: Training and validation loss during training
- **Precision, Recall, F1-Score**: Classification metrics
- **AUC (Area Under Curve)**: Classification performance
- **Custom Metrics**: Any numeric value (RMSE, MAE, etc.)

Logging Metrics

```
mlflow.log_metric('accuracy', 0.95)
mlflow.log_metric('precision', 0.92)
mlflow.log_metric('recall', 0.94)
mlflow.log_metric('f1_score', 0.93)
mlflow.log_metric('auc', 0.96)
```

5.1.2 2. Parameters

Configuration values for the run:

- **Model Hyperparameters**: Learning rate, number of trees, max depth
- **Data Processing Parameters**: Train-test split ratio, preprocessing steps
- **Feature Engineering**: Feature selection criteria, transformation parameters

Logging Parameters

```
mlflow.log_param('learning_rate', 0.01)
mlflow.log_param('n_estimators', 100)
mlflow.log_param('max_depth', 10)
mlflow.log_param('test_size', 0.2)
```

5.1.3 3. Artifacts

Files and objects associated with the run:

- **Trained Models**: Serialized model files
- **Model Summaries**: Architecture details, model info
- **Confusion Matrices**: Classification performance visualizations
- **ROC Curves**: Receiver Operating Characteristic curves
- **Plots**: Loss curves, feature importance plots

- **Input Data**: Training and testing datasets
- **Scripts & Notebooks**: Code files, Jupyter notebooks
- **Environment Files**: requirements.txt, conda.yaml

Logging Artifacts

```
# Log individual files
mlflow.log_artifact("confusion_matrix.png")
mlflow.log_artifact("model_summary.txt")
mlflow.log_artifact(__file__)  # Current script

# Log entire directory
mlflow.log_artifacts("plots/")
```

5.1.4 4. Models

Serialized models in various formats:

- **Pickled Models**: Python pickle format
- **ONNX Models**: Cross-platform ONNX format
- **Custom Models**: Using MLflow's model interface

Logging Models

```
# Log scikit-learn model
mlflow.sklearn.log_model(model, "model")

# Log TensorFlow model
mlflow.tensorflow.log_model(model, "tf-model")

# Log PyTorch model
mlflow.pytorch.log_model(model, "pytorch-model")
```

5.1.5 5. Tags

Metadata for organizing and filtering runs:

- **Run Tags**: Author name, experiment description, model type
- **Environment Tags**: GPU usage, cloud provider

Setting Tags

```
mlflow.set_tags({
    "Author": "Sujil S",
    "Project": "Wine Classification",
    "Model": "RandomForest",
    "Environment": "GPU"
})
```

5.1.6 6. Source Code

Code versioning and tracking:

- **Scripts**: Python files, notebooks
- **Git Commit**: Git commit hash
- **Dependencies**: Library versions

Logging Source Information

```
# Log current script
mlflow.log_artifact(__file__)

# Git information is automatically captured
# Dependencies can be logged with conda or requirements
```

5.1.7 7. Logging Inputs and Outputs

Data tracking:

- **Training Data**: Input datasets
- **Test Data**: Validation/test datasets
- **Inference Outputs**: Model predictions

Logging Datasets

```
import mlflow.data

# Log training data
train_df = mlflow.data.from_pandas(train_dataframe)
mlflow.log_input(train_df, "training")

# Log test data
test_df = mlflow.data.from_pandas(test_dataframe)
mlflow.log_input(test_df, "testing")
```

5.1.8 8. Custom Logging

Flexible logging for any object:

- **Custom Objects**: Any Python object
- **Custom Functions**: Track custom processing functions

5.1.9 9. Model Registry

Model lifecycle management:

- **Model Versioning**: Track different versions
- **Model Deployment**: Manage deployment status
- **Lifecycle Stages**: Staging, Production, Archived

5.1.10 10. Run and Experiment Details

Metadata automatically captured:

- **Run ID**: Unique identifier for each run
- **Experiment Name**: Logical grouping
- **Timestamps**: Start and end times

6 Understanding Tracking URI

6.1 The Tracking URI Problem

Warning

Common Error: When `mlflow.set_tracking_uri()` is not specified, artifact logging may fail with:

```
mlflow.exceptions.MlflowException: When an mlflow-artifacts
URI was supplied, the tracking URI must be a valid http or
https URI, but it was currently set to sqlite:/mlflow.db.
```

6.2 Understanding the Issue

test.py - Checking Tracking URI

```python
import mlflow

print("Printing tracking URI scheme below")
print(mlflow.get_tracking_uri())
print("\n")

mlflow.set_tracking_uri("http://127.0.0.1:5000")
print("Printing new tracking URI scheme below")
print(mlflow.get_tracking_uri())
print("\n")
```

Output:

```
Printing tracking URI scheme below
sqlite:///mlflow.db

Printing new tracking URI scheme below
http://127.0.0.1:5000
```

6.3 Why the Error Occurs

Root Cause

The error occurs because:

1. By default, MLflow uses **SQLite backend**: `sqlite:///mlflow.db`
2. Some experiments have artifact location set to **mlflow-artifacts://**
3. The `mlflow-artifacts://` scheme requires an **HTTP/HTTPS server**
4. SQLite backend is not in HTTP format
5. When logging artifacts, MLflow cannot resolve the URI

6.4 The Solution

Always set tracking URI to HTTP format:

```python
import mlflow

# Set tracking URI to MLflow server
mlflow.set_tracking_uri("http://127.0.0.1:5000")

# Now artifact logging will work
with mlflow.start_run():
    mlflow.log_artifact("plot.png")  # Works!
```

6.5 MLflow Data Storage

Important Note

All information for MLflow UI is obtained from two sources:

1. **mlflow.db**: Metadata (runs, experiments, parameters, metrics)
2. **mlartifacts/** or **mlruns/**: Actual artifacts (models, plots, files)

7 Managing Runs and Experiments

7.1 Deleting Runs

7.1.1 UI Deletion (Soft Delete)

Method: Using MLflow UI

1. Navigate to the run in UI
2. Click delete button
3. Run disappears from UI

Warning

Important: UI deletion is a **soft delete**:

- Run is NOT removed from `mlflow.db`
- Run still exists in database with `lifecycle_stage = deleted`
- It is hidden from UI but remains in the database
- Can be recovered if needed

7.1.2 Programmatic Deletion (Hard Delete)

Method: Delete directly from database

Delete_Runs.py

```python
import mlflow

# Delete run by ID
mlflow.delete_run("Run_ID")
```

```
# Run deletion script
python Delete_Runs.py
```

Effect: Deleting here removes the run from both UI and database.

7.2 Creating Custom Experiments

Instead of using the default experiment (ID: 0), create custom experiments for better organization.

7.2.1 Method 1: Using MLflow UI

1. Open MLflow UI
2. Click "Create Experiment"
3. Enter experiment name
4. Obtain Experiment ID
5. Use in code: `mlflow.start_run(experiment_id=<id>)`

main_2.py - Using Experiment ID

```python
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Load Wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# Train test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42
)

# Define the params for RF model
max_depth = 10
n_estimators = 15

# Use specific experiment by ID
with mlflow.start_run(experiment_id=1):
    rf = RandomForestClassifier(
        max_depth=max_depth,
        n_estimators=n_estimators,
        random_state=42
    )
    rf.fit(X_train, y_train)

    y_pred = rf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

    mlflow.log_metric('accuracy', accuracy)
    mlflow.log_param('max_depth', max_depth)
    mlflow.log_param('n_estimators', n_estimators)

    # Creating a confusion matrix plot
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(6,6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=wine.target_names,
                yticklabels=wine.target_names)
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title('Confusion Matrix')

    # save plot
    plt.savefig("Confusion-matrix.png")

    # log artifacts using mlflow
    mlflow.log_artifact("Confusion-matrix.png")
    mlflow.log_artifact(__file__)
```

```python

    # tags
    mlflow.set_tags({
        "Author": 'Sujil S',
        "Project": "Wine Classification"
    })

    # Log the model
    mlflow.sklearn.log_model(rf, "Random-Forest-Model")

    print(accuracy)
```

7.2.2 Method 2: Programmatic Creation

Better approach: Create experiment in code

main_3.py - Using set_experiment()

```python
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

mlflow.set_tracking_uri("http://127.0.0.1:5000")

# Load Wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# Train test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42
)

# Define the params for RF model
max_depth = 10
n_estimators = 15

# Set experiment by name (creates if doesn't exist)
mlflow.set_experiment("YT-MLOPS-Exp-2")

with mlflow.start_run():
    rf = RandomForestClassifier(
        max_depth=max_depth,
        n_estimators=n_estimators,
        random_state=42
    )
    rf.fit(X_train, y_train)

    y_pred = rf.predict(X_test)
```

```python
    accuracy = accuracy_score(y_test, y_pred)

    mlflow.log_metric('accuracy', accuracy)
    mlflow.log_param('max_depth', max_depth)
    mlflow.log_param('n_estimators', n_estimators)

    # Creating a confusion matrix plot
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(6,6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=wine.target_names,
                yticklabels=wine.target_names)
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title('Confusion Matrix')

    # save plot
    plt.savefig("Confusion-matrix.png")

    # log artifacts using mlflow
    mlflow.log_artifact("Confusion-matrix.png")
    mlflow.log_artifact(__file__)

    # tags
    mlflow.set_tags({
        "Author": 'Sujil S',
        "Project": "Wine Classification"
    })

    # Log the model
    mlflow.sklearn.log_model(rf, "Random-Forest-Model")

    print(accuracy)
```

7.3 Experiments vs Runs

Key Distinction

Experiment: A logical grouping of runs
Run: A single execution with specific parameters

Relationship:

- Each modeling approach can be an experiment
- Each parameter combination is a run within that experiment

7.3.1 Example: Wine Classification

- **Experiments**: Different models
  - Experiment 1: Random Forest
  - Experiment 2: XGBoost
  - Experiment 3: Neural Network

- **Runs**: Parameter combinations within each model
  - Random Forest Run 1: n_estimators=10, max_depth=5
  - Random Forest Run 2: n_estimators=50, max_depth=10
  - Random Forest Run 3: n_estimators=100, max_depth=15

7.3.2 Team Collaboration Scenario

- **Experiments**: Components assigned to team members
  - Member 1: Data Preprocessing Experiment
  - Member 2: Feature Engineering Experiment
  - Member 3: Model Training Experiment
- **Runs**: Different techniques tried by each member
  - Preprocessing Run 1: StandardScaler
  - Preprocessing Run 2: MinMaxScaler
  - Preprocessing Run 3: RobustScaler

8 Remote Server Setup with Dagshub

8.1 The Need for Remote Storage

Problem with Local Storage:

- Experiments stored only on local machine
- Team members cannot access each other's experiments
- No centralized view of all experiments
- Difficult to collaborate and compare results

Solution: Use remote server for centralized experiment tracking.

8.2 Remote Storage Options

8.2.1 Option 1: AWS

Setup Requirements:

1. Create IAM user
2. Set up EC2 instance for metadata storage
3. Set up S3 bucket for large files
4. Integrate MLflow with AWS

Important Note

AWS setup is powerful but time-consuming. Requires infrastructure knowledge and ongoing management.

8.2.2 Option 2: Dagshub (Recommended)

Advantages:

- No need to explicitly set up server, storage, and database
- Automatic configuration
- GitHub integration
- Free tier available
- MLflow UI built-in

8.3 Setting Up Dagshub

8.3.1 Step 1: Sign Up and Connect

1. Go to `https://dagshub.com`
2. Sign up/Sign in with GitHub
3. Connect your repository with Dagshub
4. Dagshub automatically sets up all requirements

8.3.2 Step 2: Obtain MLflow Credentials

After connecting repository, Dagshub provides:

- **MLflow Tracking URI**: Remote server URL
- **MLflow UI Link**: Shareable link for team

Example MLflow UI Link:

`https://dagshub.com/Error-Makes-Clever/MLOPS-MLflow_Experiment_Tracking.mlflow/#/experiments`

Important Note

This link can be shared with the entire team. Everyone can:

- View all experiments
- Compare results
- Review each other's work
- Access models and artifacts

8.3.3 Step 3: Install Dagshub

```
pip install dagshub
```

8.3.4 Step 4: Update Code for Remote Tracking

main_4.py - Dagshub Integration

```python
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

import dagshub

# Initialize Dagshub
dagshub.init(
    repo_owner='Error-Makes-Clever',
    repo_name='MLOPS-MLflow_Experiment_Tracking',
    mlflow=True
)

# Set tracking URI to Dagshub
mlflow.set_tracking_uri(
    "https://dagshub.com/Error-Makes-Clever/"
    "MLOPS-MLflow_Experiment_Tracking.mlflow"
)

```

```
# Load Wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# Train test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42
)

# Define the params for RF model
max_depth = 10
n_estimators = 15

mlflow.set_experiment("YT-MLOPS-Dagshub-Exp-1")

with mlflow.start_run():
    rf = RandomForestClassifier(
        max_depth=max_depth,
        n_estimators=n_estimators,
        random_state=42
    )
    rf.fit(X_train, y_train)

    y_pred = rf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

    mlflow.log_metric('accuracy', accuracy)
    mlflow.log_param('max_depth', max_depth)
    mlflow.log_param('n_estimators', n_estimators)

    # Creating a confusion matrix plot
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(6,6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=wine.target_names,
                yticklabels=wine.target_names)
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title('Confusion Matrix')

    # save plot
    plt.savefig("Confusion-matrix.png")

    # log artifacts using mlflow
    mlflow.log_artifact("Confusion-matrix.png")
    mlflow.log_artifact(__file__)

    # tags
    mlflow.set_tags({
        "Author": 'Sujil S',
        "Project": "Wine Classification"
    })

    # Log the model
    mlflow.sklearn.log_model(rf, "Random-Forest-Model")
```

```
    print(accuracy)
```

8.3.5 Step 5: Run and Verify

```
# Run the script
python src/main_4.py
```

Result:

- Experiments stored in remote server
- Visible to entire team
- No local `mlartifacts/` or `mlflow.db` created
- All data in Dagshub cloud

8.4 Team Collaboration Setup

For team members to use the same setup:

```python
import dagshub

dagshub.init(
    repo_owner='Error-Makes-Clever',
    repo_name='MLOPS-MLflow_Experiment_Tracking',
    mlflow=True
)
mlflow.set_tracking_uri(
    "https://dagshub.com/Error-Makes-Clever/"
    "MLOPS-MLflow_Experiment_Tracking.mlflow"
)
```

Key Advantage

All team members using this setup will:

- Store experiments in the same remote server
- View all experiments from all team members
- Compare and review each other's work
- Maintain centralized experiment history

9 Industry Workflow: DVC + MLflow + Git

9.1 Purpose of Each Tool

Tool	Purpose
Git	Versions code and configuration; provides rollback anchor (commit hash)
DVC	Versions data, models, and pipelines; ensures reproducibility using dvc.lock
MLflow	Tracks experiments (parameters, metrics, artifacts); helps compare runs and choose the best

Important Note

Critical Understanding:
MLflow can roll back model versions, but **not code or data** — Git and DVC handle that.

9.2 Core Industry Principle

Most Important Concept

Runs are experiments.
Commits are decisions.

You may run **many experiments**, but you commit **only the runs you decide to keep**.

9.3 Why Not Commit Before Every Run?

Reasons:

1. **Git commit hash does not exist before commit**
2. **Pollutes Git history**: Hundreds of commits for failed experiments
3. **Breaks code reviews**: Impossible to review meaningful changes
4. **Does not scale**: Unmanageable in production

Industry Standard:

- Run experiments first (exploration phase)
- Commit later (decision phase)
- Attach commit hash to the chosen run

9.4 Standard Experiment Workflow

Typical Workflow

```
# Run multiple experiments
dvc repro  # Experiment 1
dvc repro  # Experiment 2
dvc repro  # Experiment 3

# Multiple runs logged in MLflow
# No commits yet
# These runs are exploratory
```

9.5 Selecting the Best Run

In MLflow UI (Dagshub):

1. Sort runs by metric
2. Choose the best run_id

Example:

```
Best run:
  run_id = a1b2c3
  accuracy = 0.94
```

9.6 Freezing the Winning State

9.6.1 Step 1: Commit the Repository State

```
git commit -am "Freeze best model (lr=0.01, depth=5)"
```

This commit captures:

- Code
- params.yaml
- dvc.yaml
- dvc.lock

9.6.2 Step 2: Attach Commit Hash to MLflow Run

Linking MLflow Run to Git Commit

```python
import subprocess
import mlflow
from mlflow.tracking import MlflowClient
import dagshub

# Authenticate with Dagshub
dagshub.init(
    repo_owner="github_name",
    repo_name="repo_name",
```

```python
    mlflow=True
)
mlflow.set_tracking_uri("dagshub_tracking_URI")

client = MlflowClient()

run_id = "best_run_id"

# Get current Git commit hash
commit = subprocess.check_output(
    ["git", "rev-parse", "HEAD"]
).decode().strip()

# Attach commit hash to MLflow run
client.set_tag(run_id, "git_commit", commit)
```

This achieves:

- `run_id` identifies which experiment won
- `git_commit` identifies which repo state produced it
- Creates the **MLflow → Git → DVC link**

9.7 Rollback: Which Hash is Used?

Warning

Critical: Rollback always uses the **Git commit hash** stored in the MLflow tag `git_commit`

NOT:

- MLflow run ID
- MLflow artifact hashes
- DVC cache hashes

ONLY: Git commit hash

9.8 How Rollback is Performed

```
# 1. Checkout the Git commit from MLflow tag
git checkout <git_commit_from_mlflow>

# 2. Restore data using DVC
dvc pull

# 3. Reproduce the pipeline
dvc repro
```

This restores:

- Exact code

- Exact parameters
- Exact dataset version
- Exact model artifacts

9.9 What Happens to Uncommitted Runs?

Fate of Uncommitted Experiments

Status:

- They stay in MLflow
- They are useful for comparison
- They are **not rollbackable**
- This is intentional and accepted in industry

Principle: No commit means not reproducible and not deployable

9.10 What If the Best Run Was a Middle Run?

Scenario: Best run was experiment #23, but you continued to #50.
Solution:

1. Read parameters from MLflow for run #23
2. Re-run it intentionally
3. Commit immediately
4. Attach the commit hash

Important Note

This is the **correct industry behavior**. If you didn't commit it when it was best, you recreate and commit it now.

9.11 Responsibilities by Tool

Responsibility	**Tool**
Compare experiments	MLflow
Choose best run	MLflow
Rollback	Git and DVC
Data reproducibility	DVC
Deployment lifecycle	MLflow Registry

9.12 Final Mental Model

The Complete Picture

- **MLflow** tells you which run won
- The **git_commit tag** tells you what to check out
- **DVC** makes it reproducible

Workflow Summary:
You explore freely, commit intentionally, and roll back using the Git commit hash stored in MLflow.

10 MLflow Autolog Feature

10.1 What is mlflow.autolog()?

mlflow.autolog() is a powerful feature that automatically logs parameters, metrics, models, and other relevant information during machine learning training.

Autolog Benefits

- Reduces boilerplate code
- Automatically captures standard metrics
- Framework-specific intelligent logging
- Logs model artifacts automatically

10.2 What Can Be Logged by Autolog

10.2.1 1. Parameters

- Hyperparameters: max_depth, learning_rate, n_estimators
- Automatically extracted from model configuration

10.2.2 2. Metrics

- Common evaluation metrics: accuracy, precision, recall
- Loss values (for applicable frameworks)
- Framework-specific metrics

10.2.3 3. Model

- Trained model automatically logged
- Model signature inferred

10.2.4 4. Artifacts

- Model summary (if supported)
- Training plots (learning curves, confusion matrix for some frameworks)

10.2.5 5. Framework-Specific Information

- Early stopping criteria (gradient boosting)
- Number of epochs (deep learning)
- Optimizer configuration

10.2.6 6. Environment Information

- Installed libraries
- Library versions

10.2.7 7. Training Data Information

- Dataset size
- Feature information (sometimes)
- **Note**: Not the entire dataset itself

10.2.8 8. Automatic Model Signature

- Infers input types
- Saves signature with model

10.3 What Cannot Be Logged by Autolog

10.3.1 1. Custom Metrics

- Metrics not in default set for the framework
- Example: F1 score if not default
- Must be logged manually

10.3.2 2. Custom Artifacts

- Custom plots or visualizations
- Reports not part of default training

10.3.3 3. Preprocessed Data

- Transformed or preprocessed data
- Must be logged manually as artifact

10.3.4 4. Intermediate Model States

- Models saved at intermediate stages
- Checkpoints during training

10.3.5 5. Complex Model Structures

- Non-standard or highly customized models
- May miss some logging details

10.3.6 6. Non-standard Training Loops

- Custom training loops
- Not compatible with standard loops

10.3.7 7. Non-Supported Frameworks

- Frameworks without MLflow integration
- Autologging won't work

10.3.8 8. Custom Hyperparameter Tuning

- Parameters outside framework's scope
- Custom grid search configurations

10.4 Important Limitation: Pipeline Context

Warning

Critical Understanding:
MLflow autolog only logs information from scripts where an MLflow run is active.

Example:

- MLflow active only in model building script
- Only that script's details are logged
- Data ingestion runs outside MLflow context
- Feature engineering runs outside MLflow context
- Their parameters (test size, data source, feature logic) are **NOT logged automatically**

Reason: Autolog captures model-related information from supported ML libraries, not pipeline logic.

Solution: To include other script details, you must:

1. Run them under the same MLflow run, OR
2. Log their parameters manually

10.5 Autolog Usage Example

main_5.py - Using Autolog

```python
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

mlflow.set_tracking_uri("http://127.0.0.1:5000")

# Load Wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# Train test split
X_train, X_test, y_train, y_test = train_test_split(
```

```python
    X, y, test_size=0.10, random_state=42
)

# Define the params for RF model
max_depth = 10
n_estimators = 15

# Enable autologging
mlflow.autolog()
mlflow.set_experiment("YT-MLOPS-Autolog-Exp-1")

with mlflow.start_run():
    rf = RandomForestClassifier(
        max_depth=max_depth,
        n_estimators=n_estimators,
        random_state=42
    )
    rf.fit(X_train, y_train)

    y_pred = rf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

    # Creating a confusion matrix plot
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(6,6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=wine.target_names,
                yticklabels=wine.target_names)
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title('Confusion Matrix')

    # save plot
    plt.savefig("Confusion-matrix.png")

    # Manual logging still works alongside autolog
    mlflow.log_artifact(__file__)

    # tags
    mlflow.set_tags({
        "Author": 'Sujil S',
        "Project": "Wine Classification"
    })

    print(accuracy)
```

With autolog enabled:

- Parameters (max_depth, n_estimators) logged automatically
- Model logged automatically
- Training metrics logged automatically
- Manual logging (artifacts, tags) still works

10.6 Summary

Important Note

Use autolog when:

- Using supported ML frameworks
- Standard training workflows
- Want to reduce boilerplate code

Manual logging needed for:

- Custom metrics and artifacts
- Pipeline parameters from other scripts
- Complex or non-standard workflows

11 Hyperparameter Tuning with MLflow

11.1 Overview

MLflow provides excellent support for tracking hyperparameter tuning experiments. When using techniques like Grid Search or Random Search, MLflow can log:

- Parent run (orchestration)
- Child runs (individual trials)
- Best parameters and metrics
- All trial results for comparison

11.2 Complete Hyperparameter Tuning Example

main_6.py - Grid Search with MLflow

```python
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
import pandas as pd
import mlflow

# Load the Breast Cancer dataset
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target, name='target')

# Splitting into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Creating the RandomForestClassifier model
rf = RandomForestClassifier(random_state=42)

# Defining the parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [10, 50, 100],
    'max_depth': [None, 10, 20, 30]
}

# Applying GridSearchCV
grid_search = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    cv=5,
    n_jobs=-1,
    verbose=2
)

# Mention your experiment
mlflow.set_experiment('Breast-Cancer-Rf-HP-Tuning')

# Parent run for orchestration
```

```python
with mlflow.start_run() as parent:
    grid_search.fit(X_train, y_train)

    # Log all the child runs (individual trials)
    for i in range(len(grid_search.cv_results_['params'])):
        with mlflow.start_run(nested=True) as child:
            # Log parameters for this trial
            mlflow.log_params(grid_search.cv_results_["params"
    ][i])

            # Log accuracy for this trial
            mlflow.log_metric(
                "accuracy",
                grid_search.cv_results_["mean_test_score"][i]
            )

    # Get best parameters and score
    best_params = grid_search.best_params_
    best_score = grid_search.best_score_

    # Log best params to parent
    mlflow.log_params(best_params)

    # Log best metrics to parent
    mlflow.log_metric("accuracy", best_score)

    # Log training data
    train_df = X_train.copy()
    train_df['target'] = y_train
    train_df = mlflow.data.from_pandas(train_df)
    mlflow.log_input(train_df, "training")

    # Log test data
    test_df = X_test.copy()
    test_df['target'] = y_test
    test_df = mlflow.data.from_pandas(test_df)
    mlflow.log_input(test_df, "testing")

    # Log source code
    mlflow.log_artifact(__file__)

    # Log the best model
    mlflow.sklearn.log_model(
        grid_search.best_estimator_,
        "random_forest"
    )

    # Set tags
    mlflow.set_tag("author", "Sujil S")

    print(best_params)
    print(best_score)
```

11.3 Understanding Parent and Child Runs

Run Hierarchy

Parent Run: Orchestration run

- Represents the overall hyperparameter tuning process
- Contains best parameters and best score
- Links to all individual trials
- Provides high-level summary

Child Runs: Individual trials

- Each parameter combination is a separate child run
- Contains specific parameters and corresponding metrics
- Nested under parent run
- Allows detailed comparison

11.4 Comparing Runs in MLflow UI

After running the hyperparameter tuning script, you can compare all child runs:

1. Navigate to the parent run in MLflow UI
2. Select multiple child runs you want to compare
3. Click "Compare" option
4. MLflow provides various visualization options:
   - **Parallel Coordinate Plot**: Shows parameter-metric relationships
   - **Scatter Plot**: Visualizes correlation between parameters and metrics
   - **Box Plot**: Displays distribution of metrics
   - **Contour Plot**: Shows interaction effects between parameters

Important Note

The parent-child run structure provides excellent organization for hyperparameter tuning experiments, making it easy to:

- Track all trials systematically
- Compare individual parameter combinations
- Identify the best performing configuration
- Understand parameter impact on performance

12 Deep Dive: MLflow Artifact Logging Error

12.1 The Complete Error Message

Warning

```
mlflow.exceptions.MlflowException: When an mlflow-artifacts
URI was supplied, the tracking URI must be a valid http or
https URI, but it was currently set to sqlite:/mlflow.db.
```

12.2 When This Error Occurs

This error happens when **ALL** of these conditions are met:

1. Using **SQLite backend** as tracking URI (`sqlite:///mlflow.db`)
2. Experiment has artifact location set to `mlflow-artifacts://`
3. Calling `mlflow.log_artifact()` to log files
4. No **MLflow tracking server** is running

Important Note

Important: This can affect ANY experiment (default or custom), depending on when and how the experiment was created!

12.3 Specific Scenarios That Trigger the Error

Any experiment with `mlflow-artifacts://` artifact location:

- `mlflow.log_artifact("confusion-matrix.png")` ✗
- `mlflow.log_artifact(__file__)` ✗
- `mlflow.sklearn.log_model(model, "model-name")` ✗

This happens to:

- The Default experiment (most commonly affected)
- Custom experiments created when MLflow was in certain states
- Any experiment if the `mlflow.db` was created in specific ways

12.4 Why This Error Occurs

12.4.1 Root Cause

`mlflow-artifacts://` is NOT caused by SQLite itself.

When an experiment is created while MLflow is configured for a server-based deployment, it receives an artifact location of:

```
mlflow-artifacts:/0  (or mlflow-artifacts:/1, /2, etc.)
```

The `mlflow-artifacts://` URI scheme is designed for **client-server deployments** and requires:

- An HTTP/HTTPS tracking server to be running
- The server acts as a proxy to resolve artifact locations

12.4.2 Which Experiments Are Affected?

- **Default experiment (ID: 0)**: Gets `mlflow-artifacts://` if first created while MLflow was configured for a server-based artifact root
- **Custom experiments**: May or may not get `mlflow-artifacts://` depending on MLflow configuration at time of creation:
  - Created when tracking URI was HTTP-based → gets `mlflow-artifacts://`
  - Created under server defaults → gets `mlflow-artifacts://`
  - Created in pure local setup → gets `file://`

Important Note

Key Point: It's about the **creation context**, not the experiment type (default vs custom).

12.4.3 The Mismatch Problem

```
Tracking URI:     sqlite:///mlflow.db        (Local database)
Artifact URI:     mlflow-artifacts:/0        (Expects HTTP server)
                          ↓
                   INCOMPATIBLE!
```

When you try to log an artifact:

1. MLflow reads the run's artifact URI: `mlflow-artifacts:/0/...`
2. MLflow tries to resolve this URI using the tracking URI
3. It expects an HTTP/HTTPS server but finds `sqlite://`
4. **Error thrown!**

12.5 Why Custom Experiments MAY NOT Have This Problem

When you create a **custom experiment** using `mlflow.set_experiment()`:

```python
mlflow.set_experiment('My-Custom-Experiment')
```

MLflow **usually** (but not always) assigns a **file-based artifact location**:

`file:///S:/MLOPS-Lecture-5/mlruns/1`

Warning

However, this is NOT guaranteed! Custom experiments can also get `mlflow-artifacts://` URIs depending on your MLflow setup.

File-based URIs work perfectly with SQLite backend:

```
Tracking URI:     sqlite:///mlflow.db        (Local database)
Artifact URI:     file:///path/to/mlruns/1   (Local filesystem)
                          ↓
                   COMPATIBLE!
```

12.6 How to Check If Your Experiment Is Affected

Run this diagnostic for ANY experiment:

Diagnostic Script

```python
import mlflow
from mlflow.tracking import MlflowClient

client = MlflowClient()

# Check specific experiment by name
exp = client.get_experiment_by_name('Your-Experiment-Name')
print(f"Artifact Location: {exp.artifact_location}")

# If it shows "mlflow-artifacts://..." -> You'll get the error!
# If it shows "file://..." -> You're safe!
```

12.7 Solutions

12.7.1 Solution 1: Use Fresh Custom Experiment

Recommended for Local Development

Creating New Experiment

```python
import mlflow

# Create a NEW experiment (if it doesn't exist yet)
mlflow.set_experiment('Wine-Classification-v2')

with mlflow.start_run():
    # Your code...
    mlflow.log_artifact("plot.png")  # Should work if file://
    location
```

Warning

Important: This only works if the newly created experiment gets a `file://` artifact location. Always verify!

Verification:

```python
from mlflow.tracking import MlflowClient
client = MlflowClient()
exp = client.get_experiment_by_name('Wine-Classification-v2')
print(f"Artifact location: {exp.artifact_location}")
# Make sure it says "file://" not "mlflow-artifacts://"
```

Pros:

- Simple if it works
- No external dependencies
- Works offline

Cons:

- NOT guaranteed to work - depends on MLflow configuration
- Need to verify artifact location
- May still get `mlflow-artifacts://` URI

12.7.2 Solution 2: Run MLflow Tracking Server

Recommended for Production

Step 1: Start MLflow server in a separate terminal

```
mlflow server --host 127.0.0.1 --port 5000
```

Step 2: Point your script to the server

```
import mlflow

mlflow.set_tracking_uri("http://127.0.0.1:5000")

with mlflow.start_run():
    # Your code...
    mlflow.log_artifact("plot.png")  # Works!
```

Pros:

- Works with default experiment
- Production-ready setup
- Better for team collaboration
- Web UI accessible from any network location

Cons:

- Need to keep server running
- Extra setup step

12.7.3 Solution 3: Set Environment Variable

```
# Windows PowerShell
$env:MLFLOW_TRACKING_URI = "http://127.0.0.1:5000"

# Windows CMD
set MLFLOW_TRACKING_URI=http://127.0.0.1:5000

# Linux/Mac
export MLFLOW_TRACKING_URI=http://127.0.0.1:5000
```

Then run your script without any code changes.

Pros:

- No code changes needed
- Consistent across all scripts

Cons:

- Environment-dependent
- Still need MLflow server running

12.7.4 Solution 4: Use Local File Tracking

Most Reliable for Local Development

```python
import mlflow

# Use file-based tracking instead of SQLite
mlflow.set_tracking_uri("file:///S:/MLOPS-Lecture-5/mlruns")
# or simply
mlflow.set_tracking_uri("./mlruns")

with mlflow.start_run():
    # Your code...
    mlflow.log_artifact("plot.png")  # Works reliably!
```

Pros:

- No server needed
- Works with any experiment (default or custom)
- Most reliable for local development
- Guaranteed to avoid `mlflow-artifacts://` issues

Cons:

- Different tracking backend
- Less structured than SQLite
- Need to set this in every script

12.8 Comparison Table

Aspect	mlflow-artifacts://	file://
Artifact Location	mlflow-artifacts:/0	file:///path/to/mlruns/N
Works with SQLite?	✗ No	✓ Yes
Needs HTTP Server?	✓ Yes	✗ No
Commonly Affects	Default experiment, some custom	Newly created custom (usually)
Best Fix	Use MLflow server	Already works!

12.9 Quick Diagnostic

Check if YOUR experiment will cause this error:

Complete Diagnostic

```python
import mlflow
from mlflow.tracking import MlflowClient

print(f"MLflow version: {mlflow.__version__}")
print(f"Tracking URI: {mlflow.get_tracking_uri()}")

client = MlflowClient()

# Check default experiment
exp = client.get_experiment("0")
print(f"\nDefault experiment artifact location: {exp.
    artifact_location}")

# Check YOUR custom experiment
exp = client.get_experiment_by_name('Your-Experiment-Name')
if exp:
    print(f"Your experiment artifact location: {exp.
    artifact_location}")

# If you see "mlflow-artifacts://...", you'll get the error
    with SQLite!
# If you see "file://...", you're safe!
```

13 MLflow Model Registry

13.1 What is a Model Registry?

A **Model Registry** is a central catalog where you store, version, track, and manage machine learning models across their entire lifecycle — from experimentation to retirement.

Model Registry Concept

Think of it as **Git + release management for ML models**, with extra information about:

- Which model is active
- Which one is being tested
- Which one is retired
- Why decisions were made

13.2 Why Model Registry Is Needed

13.2.1 Without Model Registry

- Don't know which model made which decision
- Can't explain past predictions
- Risk compliance violations
- Rollbacks become painful or impossible

13.2.2 With Model Registry

- Every model is traceable
- Decisions are explainable
- Changes are auditable
- Rollbacks are instant

13.3 Model Lifecycle Stages

MLflow provides a Model Registry feature with four official lifecycle stages:

Stage	**Meaning**
None	Default stage; model is registered but not promoted yet
Staging	Model is under testing or validation
Production	Model actively serving real-world predictions
Archived	Model is retired but kept for history

Important Note

Important Clarifications:

- "Development" is NOT a stage name in MLflow

- MLflow uses **None** instead of "Development"
- "Archive" → Correct term is **Archived**
- "Stagging" → Correct spelling is **Staging**

13.3.1 1. None (Default)

- Model is trained and evaluated
- Logged as an artifact
- Not yet approved for use
- Many models can exist here

Example: Data scientist experiments with features or algorithms

13.3.2 2. Staging

- Model passed initial evaluation
- Used for testing, A/B tests, or shadow deployment
- Compared against the production model

Example: New model tested on real traffic but decisions are not applied

13.3.3 3. Production

- Model actively used in real-world decisions
- Directly impacts users or business outcomes
- Only one (or a few) models should be here

Example: Model deciding loan approvals or recommendations

13.3.4 4. Archived (Retirement)

- Model is no longer active
- NOT deleted
- Preserved for:
  - Audits
  - Explanations
  - Legal compliance
  - Historical analysis

Key Understanding

Retirement ≠ Deletion ✓
Retirement = Archiving ✓

Archived models are preserved for accountability and compliance.

13.4 Real-World Example: Credit Card Fraud Detection

Timeline of Models

Background: A bank uses ML models to detect fraudulent transactions.

2022 — Model v1 (Rule-based + Logistic Regression)

- High false positives
- Many genuine transactions blocked
- Eventually retired

2023 — Model v2 (Random Forest)

- Better accuracy
- Deployed to Production
- Stored in Model Registry

2024 — Model v3 (XGBoost)

- Higher precision
- Lower customer complaints
- Moved to Staging, then Production
- Model v2 moved to Archived

13.4.1 Incident in 2025: Why Registry Matters

A customer files a complaint:

> "Why was my transaction declined on **March 14, 2023**?"

The bank must answer:

- Which model made the decision?
- What logic was used at that time?
- Was it compliant with regulations?

With Model Registry:

The bank checks the Model Registry and finds:

- Model v2 was in Production on that date
- Training data version
- Features used
- Thresholds

- Evaluation metrics

Result:

- ✓ Legal compliance
- ✓ Customer trust
- ✓ No guesswork
- Bank reproduces the decision and explains it to regulator

Without Model Registry:

- Model files overwritten
- No version history
- No explanation possible
- ✗ Regulatory violation
- ✗ Heavy penalties

13.5 Key Insight

> **Warning**
>
> **Most Important Understanding**:
> The Model Registry is not about model storage — it's about **decision accountability over time**.

13.6 How Model Registration Works in MLflow

13.6.1 Registration Process

1. **Select a logged model**:
   - From an experiment run
   - Usually from Artifacts → model section
2. **Click "Register Model"**
3. MLflow asks:
   - **Register as a new model**: First time this model name is used
   - **Register as a new version of existing model**: Model already exists in registry
4. MLflow creates:
   - A registered model name
   - A new model version (v1, v2, v3, ...)
5. (Optional) Assign a stage to the model version

13.6.2 What MLflow Model Registry Shows

For each model version, MLflow stores:

- Version number (v1, v2, v3...)
- Current stage
- Run ID that created it
- Metrics & parameters
- Timestamp
- Transition history (who moved it, when)

This gives **full traceability**.

13.7 Model Registry Operations

13.7.1 After Registration

Once registered, you can:

- See which model version is in which stage
- Promote (Staging → Production)
- Roll back (Production → Archived)
- Compare versions
- Track history of transitions

Structure

✓ One model name
✓ Multiple versions
✓ Only selected versions are Production

13.8 Simple Analogy

Software	**ML Equivalent**
Git tags	Model versions
Release branches	Staging / Production
Deprecated API	Retired model
Rollback	Promote older model

13.9 Why Archived Models Are Kept

Reasons for keeping retired models:

1. **Regulatory audits**: Finance, healthcare sectors
2. **Legal disputes**: Prove decisions were made correctly
3. **Post-mortem analysis**: Understand what went wrong
4. **Explainability requirements**: Regulatory compliance
5. **Historical comparison**: Compare new models against old

13.10 One-Line Summary

Memorize This

A Model Registry is a system that tracks which ML model version was used, when, why, and in what stage — ensuring traceability, governance, and explainability.

14 MLflow Best Practices

14.1 Experiment Organization

14.1.1 Naming Conventions

- **Experiments**: Use descriptive names
  - Good: "Wine-Classification-RandomForest"
  - Good: "Fraud-Detection-XGBoost-v2"
  - Bad: "Experiment1", "Test"
- **Runs**: Use meaningful tags
  - Tag with author, date, purpose
  - Include model type and key parameters

14.1.2 Logical Grouping

- One experiment per model type or approach
- Group related runs together
- Use tags for additional organization

14.2 What to Log

14.2.1 Always Log

- **All hyperparameters**: Even if using defaults
- **Multiple metrics**: Don't just log accuracy
- **Training data info**: Size, version, source
- **Environment details**: Python version, library versions
- **Model artifacts**: Always save the trained model
- **Source code**: Log the script used

14.2.2 Consider Logging

- Confusion matrices and plots
- Feature importance
- Training time
- Memory usage
- Dataset checksums

14.3 Tracking URI Best Practices

- **Always set explicitly**: Don't rely on defaults
- **Use HTTP format**: For server-based tracking
- **Set at script start**: Before any MLflow calls
- **Consistent across team**: Everyone uses same URI

Best Practice Template

```python
import mlflow

# Set tracking URI first thing
mlflow.set_tracking_uri("http://127.0.0.1:5000")

# Then proceed with experiments
mlflow.set_experiment("My-Experiment")

with mlflow.start_run():
    # Your code here
    pass
```

14.4 Model Registry Best Practices

1. **Clear versioning strategy**:
   - Register all candidate models
   - Use meaningful model names
   - Document version changes
2. **Stage transitions**:
   - Test thoroughly in Staging
   - Only one model in Production (usually)
   - Archive instead of delete
3. **Documentation**:
   - Add descriptions to models
   - Document stage transition reasons
   - Link to relevant runs

14.5 Team Collaboration

- **Use remote tracking**: Dagshub or MLflow server
- **Consistent naming**: Agree on naming conventions
- **Tag ownership**: Use author tags
- **Regular reviews**: Discuss experiments as team
- **Clean up old runs**: Archive or delete obsolete experiments

14.6 Common Mistakes to Avoid

Warning

Don't:

- Use default experiment for everything
- Forget to log important parameters
- Mix tracking URIs in same project
- Delete models instead of archiving
- Log sensitive data or credentials
- Commit without linking to MLflow run
- Use vague experiment names
- Skip model registration for production models

14.7 Integration with DVC and Git

1. **Run experiments first**: Explore with MLflow
2. **Choose best run**: Use MLflow UI to compare
3. **Commit decision**: Git commit with meaningful message
4. **Link commits**: Tag MLflow run with Git commit hash
5. **Push data**: Use DVC push for data versioning

15 Quick Reference Guide

15.1 Essential MLflow Commands

Installation & Setup

```
# Install MLflow
pip install mlflow

# Start MLflow UI
mlflow ui

# Start MLflow server
mlflow server --host 127.0.0.1 --port 5000

# Install Dagshub integration
pip install dagshub
```

Basic Tracking

```
import mlflow

# Set tracking URI
mlflow.set_tracking_uri("http://127.0.0.1:5000")

# Set experiment
mlflow.set_experiment("experiment-name")

# Start run
with mlflow.start_run():
    # Log parameters
    mlflow.log_param("param_name", value)

    # Log metrics
    mlflow.log_metric("metric_name", value)

    # Log artifacts
    mlflow.log_artifact("file.png")

    # Log model
    mlflow.sklearn.log_model(model, "model-name")

    # Set tags
    mlflow.set_tags({"key": "value"})
```

Autolog

```
import mlflow

# Enable autologging
mlflow.autolog()

# Train model - automatically logs parameters, metrics, model
model.fit(X_train, y_train)
```

Nested Runs for Hyperparameter Tuning

```
import mlflow

# Parent run
with mlflow.start_run() as parent:
    # Your grid search code

    # Child runs for each trial
    for params in param_combinations:
        with mlflow.start_run(nested=True) as child:
            mlflow.log_params(params)
            mlflow.log_metric("accuracy", score)

    # Log best to parent
    mlflow.log_params(best_params)
    mlflow.log_metric("best_accuracy", best_score)
```

Dagshub Integration

```
import dagshub
import mlflow

# Initialize Dagshub
dagshub.init(
    repo_owner='username',
    repo_name='repo-name',
    mlflow=True
)

# Set tracking URI
mlflow.set_tracking_uri("https://dagshub.com/username/repo.mlflow")

# Rest of your code...
```

15.2 Model Registry Commands

Model Registry Operations

```
from mlflow.tracking import MlflowClient

client = MlflowClient()

# Register model
result = mlflow.register_model(
    model_uri="runs:/<run_id>/model",
    name="model-name"
)

# Transition model stage
client.transition_model_version_stage(
    name="model-name",
    version=1,
    stage="Production"
)

# Get latest model version
latest_version = client.get_latest_versions(
    name="model-name",
    stages=["Production"]
)

# Archive model
client.transition_model_version_stage(
    name="model-name",
    version=1,
    stage="Archived"
)
```

15.3 Git Integration

Linking MLflow to Git

```
import subprocess
from mlflow.tracking import MlflowClient

client = MlflowClient()

# Get Git commit hash
commit = subprocess.check_output(
    ["git", "rev-parse", "HEAD"]
).decode().strip()

# Attach to MLflow run
client.set_tag(run_id, "git_commit", commit)

# Later: Rollback using commit hash
# git checkout <commit_from_mlflow>
# dvc pull
# dvc repro
```

15.4 Common Code Patterns

15.4.1 Complete Experiment Template

Full Experiment Template

```python
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score

# Setup
mlflow.set_tracking_uri("http://127.0.0.1:5000")
mlflow.set_experiment("My-Experiment")

# Load data
X_train, X_test, y_train, y_test = load_and_split_data()

# Parameters
params = {
    'n_estimators': 100,
    'max_depth': 10,
    'random_state': 42
}

# Train and log
with mlflow.start_run():
    # Train model
    model = RandomForestClassifier(**params)
    model.fit(X_train, y_train)

```

```
    # Predictions
    y_pred = model.predict(X_test)

    # Calculate metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')

    # Log everything
    mlflow.log_params(params)
    mlflow.log_metric("accuracy", accuracy)
    mlflow.log_metric("precision", precision)
    mlflow.sklearn.log_model(model, "model")
    mlflow.set_tags({"author": "Your Name", "project": "Project Name"})

    print(f"Accuracy: {accuracy:.4f}")
```

15.4.2 Dagshub Complete Example

Dagshub Integration Template

```
import mlflow
import mlflow.sklearn
import dagshub

# Initialize Dagshub
dagshub.init(
    repo_owner='your-username',
    repo_name='your-repo',
    mlflow=True
)

# Set tracking URI
mlflow.set_tracking_uri(
    "https://dagshub.com/your-username/your-repo.mlflow"
)

# Set experiment
mlflow.set_experiment("Remote-Experiment")

# Your experiment code
with mlflow.start_run():
    # Train model
    model = train_model()

    # Log everything
    mlflow.log_params(params)
    mlflow.log_metrics(metrics)
    mlflow.sklearn.log_model(model, "model")

    # All data stored remotely!
```

15.5 Troubleshooting Checklist

Issue	**Solution**
Artifact logging fails	Set tracking URI to HTTP format
Experiments not visible to team	Use Dagshub or MLflow server
Cannot find old experiment	Check experiment ID/name spelling
Model not in registry	Register model from run artifacts
Slow logging	Check network/server connection
Run deleted accidentally	Use programmatic deletion for recovery
Wrong experiment	Use set_experiment() before start_run()

15.6 MLflow UI Shortcuts

- **Compare runs**: Select multiple runs → Click "Compare"
- **Filter experiments**: Use search bar with filters
- **Sort by metric**: Click column header
- **View artifacts**: Click run → Artifacts tab
- **Register model**: Artifacts → model → Register Model
- **Delete run**: Three dots menu → Delete (soft delete)

16 Comparison Tables

16.1 DVC vs MLflow Summary

Feature	DVC	MLflow
Primary Purpose	Data versioning	Experiment tracking
Git Dependency	Required	Optional
UI Quality	Basic	Rich and intuitive
Team Collaboration	Local only	Centralized
Maturity (Experiments)	Newer	More mature
Data Versioning	Excellent	Basic
Model Registry	No	Yes
Visualization	Limited	Extensive

16.2 Storage Options Comparison

Setup	Pros	Cons	Best For
Local (SQLite)	Simple, offline	No collaboration	Solo projects
MLflow Server	Full features, team access	Requires server	Small teams
Dagshub	Easy setup, free tier	Internet required	Teams, learning
AWS	Scalable, production	Complex setup	Production

16.3 Logging Methods Comparison

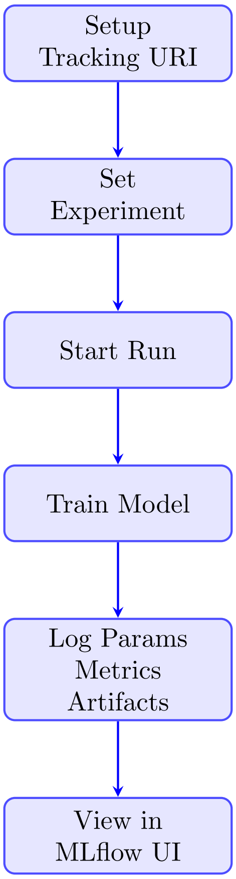
Method	Use When	Limitations
Manual Logging	Custom metrics, full control	More code
Autolog	Standard workflows, quick setup	Framework-specific
Callbacks	Deep learning training	Framework-dependent

16.4 Model Lifecycle Stages

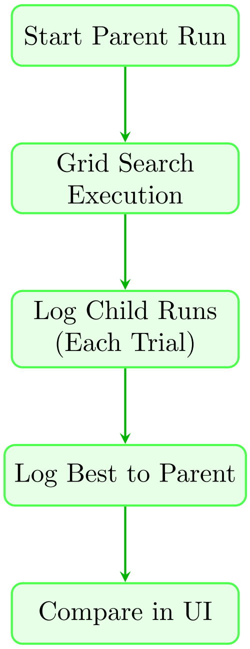
Stage	Purpose	Action
None	Initial registration	Evaluate and test
Staging	Testing phase	Compare with production
Production	Active deployment	Monitor performance
Archived	Retired	Keep for audit/compliance

17 Complete Workflow Visualizations

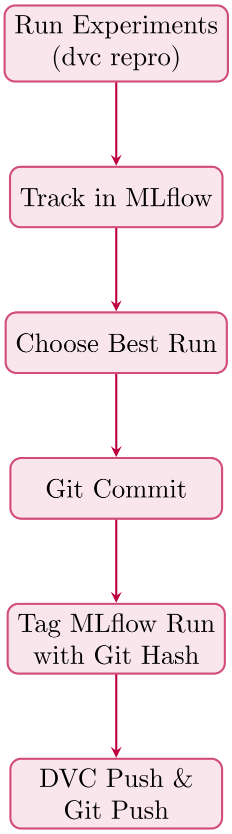
17.1 Basic MLflow Workflow



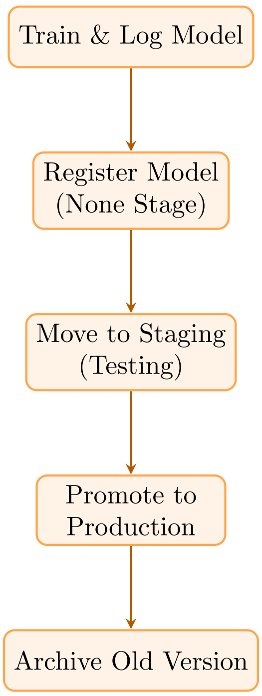
17.2 Hyperparameter Tuning Workflow



17.3 DVC + MLflow + Git Integration



17.4 Model Registry Workflow



18 Conclusion and Key Takeaways

18.1 What You've Learned

Throughout this comprehensive guide, you've mastered:

1. **MLflow Fundamentals**:
   - Installation and setup
   - Tracking URI configuration
   - Experiment and run management
2. **Experiment Tracking**:
   - Manual logging (parameters, metrics, artifacts)
   - Autolog for automated tracking
   - Nested runs for hyperparameter tuning
3. **Remote Collaboration**:
   - Dagshub integration
   - Team-wide experiment sharing
   - Centralized tracking server
4. **DVC + MLflow + Git Integration**:
   - Industry-standard workflow
   - Runs vs commits distinction
   - Proper rollback procedures
5. **Model Registry**:
   - Model lifecycle management
   - Stage transitions
   - Compliance and auditability

18.2 Core Principles to Remember

The Five Pillars of MLflow

1. **Track Everything**: Parameters, metrics, artifacts, code
2. **Organize Logically**: Experiments group related runs
3. **Collaborate Effectively**: Use remote servers for team work
4. **Commit Intentionally**: Runs are experiments, commits are decisions
5. **Maintain Accountability**: Model registry for compliance

18.3 MLflow vs DVC: When to Use Each

Use MLflow For	Use DVC For
Experiment tracking and comparison	Data and model versioning
Model registry and lifecycle	Pipeline automation
Team collaboration on experiments	Reproducible workflows
Hyperparameter tuning tracking	Large file management
Model deployment decisions	Git-style data version control

Important Note

Best Practice: Use BOTH together!

- DVC for data/pipeline versioning
- MLflow for experiment tracking
- Git for code versioning
- Together: Complete MLOps solution

18.4 The Complete MLOps Stack

Git — Code Versioning

DVC — Data & Pipeline Versioning

MLflow — Experiment Tracking & Model Registry

Deployment — Production Serving

18.5 Industry Workflow Recap

The Complete Industry Process

1. **Experiment Phase**:
   - Run multiple experiments with MLflow tracking
   - Compare results in MLflow UI
   - Iterate on parameters and approaches
2. **Decision Phase**:
   - Choose best run from MLflow
   - Git commit the winning configuration
   - Tag MLflow run with Git commit hash
3. **Deployment Phase**:
   - Register model in MLflow Registry

- Move through stages: None → Staging → Production
- Monitor and maintain

4. **Maintenance Phase**:
   - Archive old models (don't delete)
   - Maintain audit trail
   - Rollback using Git commit hash when needed

18.6 Critical Reminders

Warning

Never Forget:

1. Always set tracking URI explicitly
2. Runs are experiments, commits are decisions
3. Archived $\neq$ Deleted (keep for compliance)
4. Git commit hash is the rollback anchor
5. MLflow tracks experiments, but Git + DVC enable rollback
6. Team collaboration requires remote server (Dagshub/MLflow server)
7. Log enough information to reproduce results
8. Register production models in Model Registry

18.7 Next Steps

To continue your MLflow journey:

1. **Practice**: Set up MLflow in your own projects
2. **Experiment**: Try different tracking configurations
3. **Collaborate**: Set up Dagshub for team projects
4. **Integrate**: Combine DVC + MLflow + Git workflow
5. **Explore**: Advanced features like MLflow Projects, Model Serving
6. **Contribute**: Join the MLflow community

18.8 Additional Resources

- Official MLflow Documentation: `https://mlflow.org/docs/latest/index.html`
- MLflow GitHub Repository: `https://github.com/mlflow/mlflow`
- Dagshub Platform: `https://dagshub.com`

- MLflow Tutorials: `https://mlflow.org/docs/latest/tutorials-and-examples/index.html`
- DVC + MLflow Integration: `https://dvc.org/doc/use-cases/versioning-data-and-model-files`

18.9 Final Thoughts

MLflow has become an essential tool in the modern ML engineer's toolkit. By providing:

- **Experiment tracking** that scales from solo projects to large teams
- **Model registry** that ensures governance and compliance
- **Integration capabilities** that work with existing tools
- **Flexibility** to adapt to various workflows

MLflow enables you to move from experimental notebooks to production-ready ML systems with confidence.

The Golden Rule of MLOps

Track everything, commit intentionally, deploy confidently.

With MLflow, DVC, and Git working together, you have complete control over your ML lifecycle — from the first experiment to the last model retirement.

End of MLflow Complete Guide

"Without data, you're just another person with an opinion."
— W. Edwards Deming

With MLflow, your data becomes actionable insights!