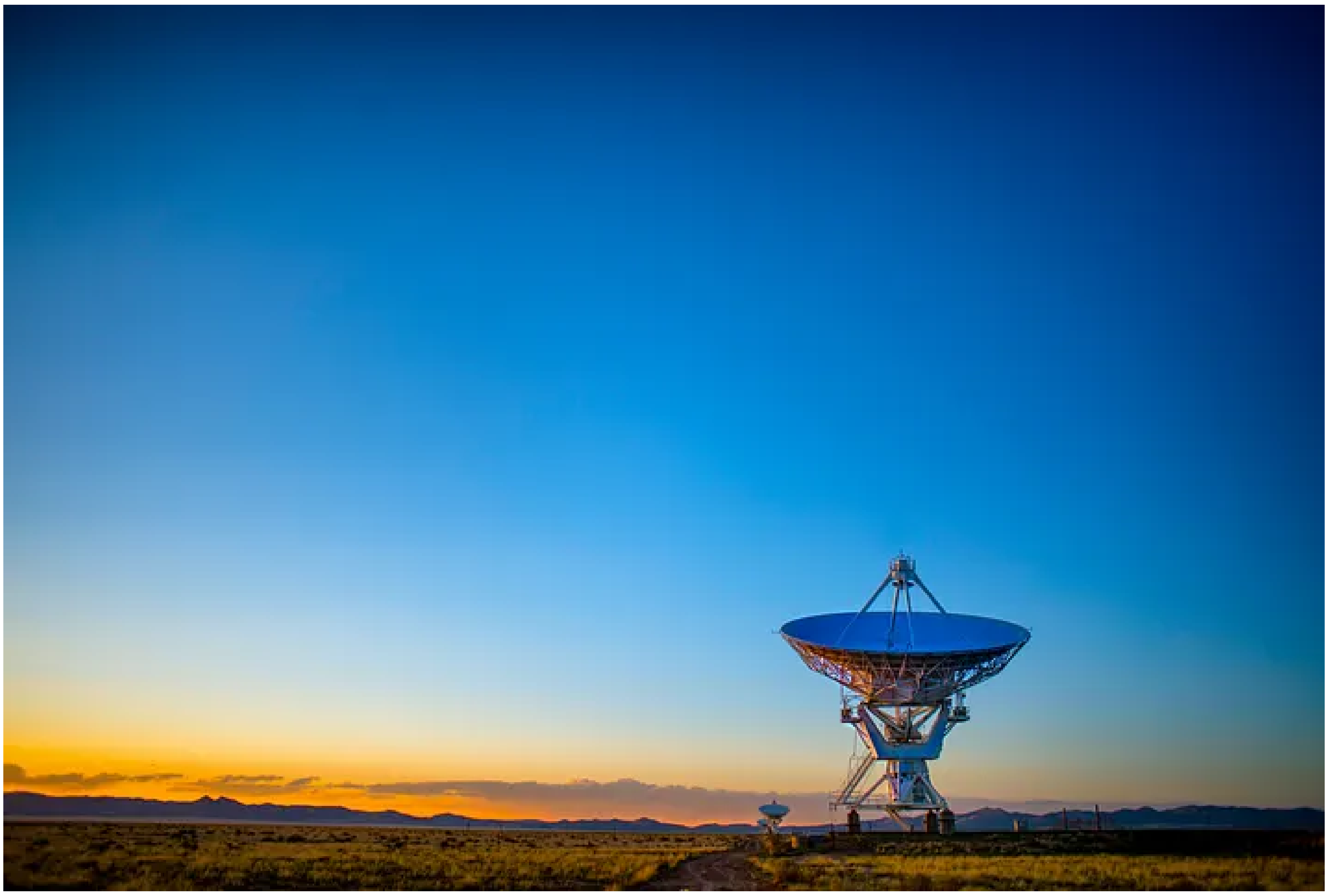
End-to-End Machine Learning Deployment with MLflow and Automation

A Step-by-Step Guide to Deploying ML Models with MLflow and CI/CD

15 min read · Feb 22, 2025

Rico Febrian Follow



Source

Hello, and welcome (again) to my learning logs!

As I continue exploring data engineering, I've recently been learning about **MLOps (Machine Learning Operations)**.

This was my very first-time exploring machine learning. I didn't even know what a "machine learning model" was when I started. But as I learned more, things slowly started to make sense.

Now, in this article, I'll Walk you through how I deployed a machine learning model using **MLflow** and automated the process with **CI/CD pipelines.**

Whether you're new to machine learning or just curious about MLOps, this guide is for you. I'll keep things simple and easy to follow, so you can learn alongside me. Let's get started!

. . .

Objective

This project isn't about achieving the best model performance. Instead, my goal is to explore and understand the end-to-end machine learning lifecycle.

By the end of this guide, I hope you'll understand how all the parts of the process fit together. Here's what we'll cover:

- **Set up cloud infrastructure using AWS EC2.**
- **Track and store code and data with Git, DVC, GitHub, and MinIO.**
- **Build a machine learning workflow using MLflow.**
- **Perform unit testing.**
- **Deploy the model with CI/CD using GitHub Actions.**
- **Run load testing with Locust.**

. . .

What You'll Need

Before we begin, this guide covers topics like Linux, Git, cloud services, and Docker. If you're unfamiliar with these, it's a good idea to learn a bit about them first so you can follow the guide without confusion.

. . .

Dataset Overview

I used a dataset for classifying cars into categories based on their **acceptability level**:

- `unacc` : Unacceptable
- `acc` : Acceptable
- `good` : Good
- `vgood` : Very good

Each car is described by six key attributes:

- **buying price:** `vhigh` , `high` , `med` , `low`
- **maintenance cost:** `vhigh` , `high` , `med` , `low`
- **number of doors:** `2` , `3` , `4` , `5more`
- **capacity in terms of persons:** `2` , `4` , `more`
- **size of luggage boot:** `small` , `med` , `big`
- **safety level:** `low` , `med` , `high`

You can check out the full dataset **here**.

. . .

Alright let's begin! Here are the steps I followed

Set Up Infrastructure in AWS EC2

Step 1: Create Instances

First, choose a cloud provider that fits your needs. For this project, I used Amazon EC2. I created six instances using Ubuntu as the operating system. Here's what each instance is used for:

MLflow Server

- Runs the MLflow service to manage the machine learning lifecycle.
- Stores model files (artifacts) in **MinIO**.
- Saves model metadata in **PostgreSQL**.

Web Server

- Runs **NGINX** to manage incoming user traffic.
- Uses a **load balancer** and **reverse proxy** to route requests to the correct environment (staging or production).

Staging Servers (Staging 1 and 2)

- Used for testing and development to check if everything works before moving to production.

Production Servers (Production 1 and 2)

- Host the final model for end users.
- Users access the model through these servers.

—

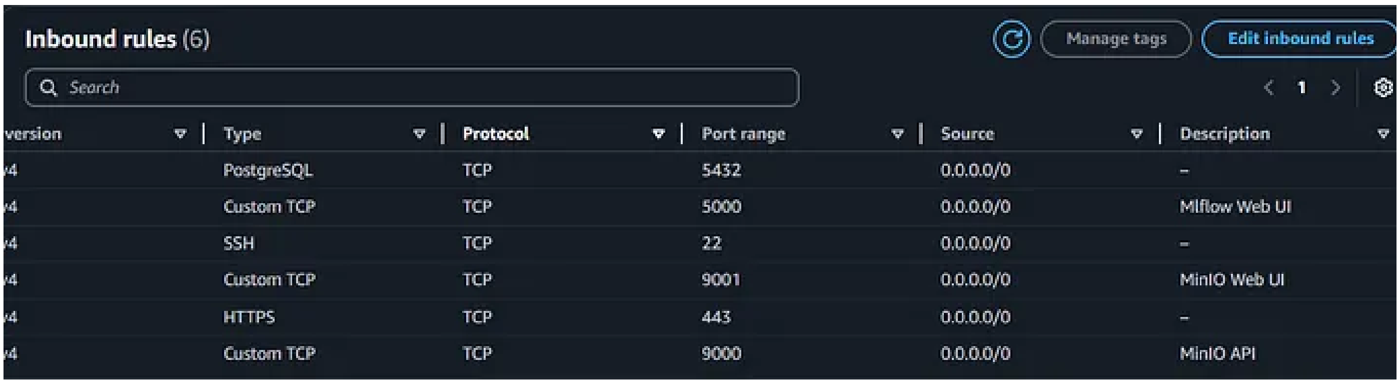
Step 2: Set Up the Security Group

After launching your instances, configure the **security group** to manage inbound and outbound traffic. This ensures your servers can receive necessary traffic for web interfaces, APIs, or other external interactions.

Many services use specific default ports. For example:

- MLflow typically uses port **5000**, so ensure this port is open.
- Similarly, check and open the ports required by other services like MinIO or custom APIs.

Below are the ports I configured:



Inbound rules (6)

version	Type	Protocol	Port range	Source	Description
v4	PostgreSQL	TCP	5432	0.0.0.0/0	–
v4	Custom TCP	TCP	5000	0.0.0.0/0	Mlflow Web UI
v4	SSH	TCP	22	0.0.0.0/0	–
v4	Custom TCP	TCP	9001	0.0.0.0/0	MinIO Web UI
v4	HTTPS	TCP	443	0.0.0.0/0	–
v4	Custom TCP	TCP	9000	0.0.0.0/0	MinIO API

Security Group Settings

—

Step 3: Set Up SSH Configuration (Optional)

To simplify SSH connections, configure the `~/.ssh/config` file. For example:

```
Host mlflow-server
    HostName 192.168.1.11
    User ubuntu
    IdentityFile ~/.ssh/my-keypair.pem
```

Ensure your key file has the correct permissions:

```
chmod 600 ~/.ssh/my-keypair.pem
```

You can now connect using:

```
ssh mlflow-server
```

. . .

Install Docker in Selected Instances

Once all servers (Staging, Production, and MLflow Server) are accessible, install Docker on each instance. This is required because the machine learning model and MLflow service will run within Docker containers.

Step 1: SSH Into Each Server

First, connect to each server using SSH.

—

Step 2: Install Docker

You can either create a script or run the following command from this **script.**

—

Step 3: Run Docker Without Sudo (Optional)

To run Docker commands without `sudo`, execute:

```
sudo usermod -aG docker $USER && sudo service docker restart
```

Then, log out and log back in to apply the changes.

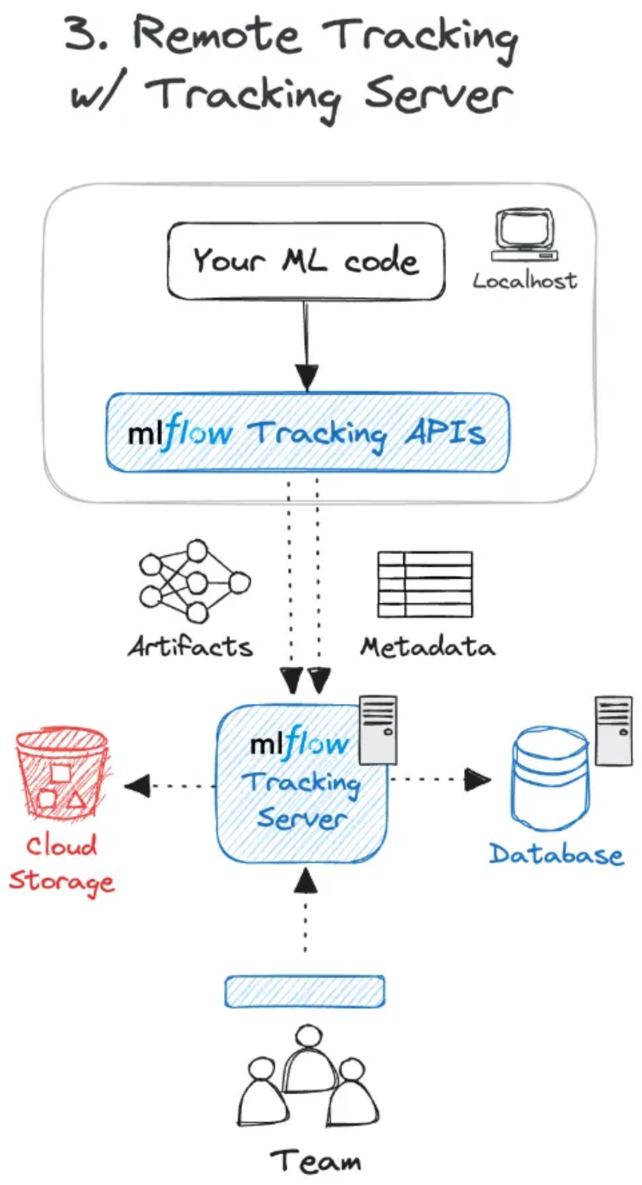
. . .

Set Up MLflow Server

I used the MLflow remote tracking model in this project, so the MLflow server runs the following services:

- **MLflow**: Tracks experiments and manages models.
- **MinIO**: Stores MLflow artifacts and DVC data.
- **PostgreSQL**: Stores MLflow metadata (e.g., experiment and run details) and authentication data.

Here’s what the MLflow remote tracking model looks like:



MLflow Remote Tracking Model

. . .

Step 1: Set Up MLflow Service

Follow these steps to set it up:

SSH into MLflow and create directories for each service

```
mkdir mlflow minio postgres
```

—

Set up the MLflow Dockerfile

```
cd mlflow && nano Dockerfile
```

Add this content:

```
FROM python:3.13.1-slim

RUN apt update && apt upgrade -y && apt install -y curl
RUN pip install mlflow[extras] psycopg2-binary boto3 cryptography pymysql

EXPOSE 5000
```

This file sets up a Python environment with MLflow and its dependencies. Port `5000` is exposed so you can access the MLflow web interface.

—

Set up MLflow authentication

To secure the MLflow web interface, you can create an authentication configuration file. Inside the `mlflow/` directory, create a folder named `config` and add a file `mlflow-config.ini`:

```
mkdir config && cd config && nano mlflow-config.ini
```

Add this content:

```
[mlflow]
default_permission=READ
database_uri=postgresql://postgres:<postgres_password>@<postgres_username>:<postgres_port>/<database_name_for_mlflow_auth>
admin_username= define_any_username
admin_password= define_any_password
```

Replace placeholders (`postgres_password`, `postgres_username`, etc.) with actual credentials.

This configuration ensures that only users with the correct username can access the MLflow web interface. Without these credentials, access will be denied.

. . .

Step 2: Set Up MinIO Service

Follow these steps to set it up:

Create required directories

```
cd minio && mkdir certs data
```

- **`certs`**: Stores SSL self-signed certificate data.
- **`data`**: Stores files and acts as a bucket.

—

Generate a self-signed certificate

To secure your MinIO service, you'll need an SSL certificate:

- Download **certgen** for your specific OS and platform **here**.
- Place the file to the `minio/certs` directory.
- Make the file executable:

```
chmod 700 certgen_name
```

- Generate the certificate:

```
./certgen_name -ed25519 -host "localhost"
```

—

Create pre-defined MinIO buckets

You can create buckets (storage folders) in the MinIO web interface later, or pre-define them now by creating two folders in `minio/data`:

```
cd minio/data && mkdir dvc mlflow
```

- **`dvc`** : Stores data managed by DVC.
- **`mlflow`** : Stores MLflow artifacts (model files and outputs).

. . .

Step 3: Set Up PostgreSQL Service
Follow these steps to set it up:

Create required directories

```
cd postgres && mkdir init data
```

- **`init`** : Stores pre-defined SQL queries that run when the PostgreSQL service starts.
- **`data`** : Stores the database files (metadata).

—

Create pre-defined queries

```
cd init && nano init.sql
```

Add this content:

```
create database mlflow;
create database mlflow_users;
```

The `mlflow` database stores metadata about MLflow experiments and related data, while the `mlflow_users` database stores authentication data for MLflow users.

. . .

Step 4: Initialize Docker Compose

This Docker Compose setup defines and runs all services (MLflow, MinIO, PostgreSQL). Follow these steps to set it up:

Create a `.env` file

- In the root project directory, create a `.env` file. Refer to this **script** for the configuration.
- Secure the `.env` file by running `chmod 600 .env`

Create Docker Compose

- In the root project directory, create a `docker-compose.yaml` file. See the full configuration **here**.

Run Docker Compose

- Start all services by running `docker compose up -d`

Verify services

- Open your browser and go to the URLs provided by the services to confirm they are working.

Check out my MLflow server infrastructure **here**.

. . .

Set Up Web Server (NGINX)

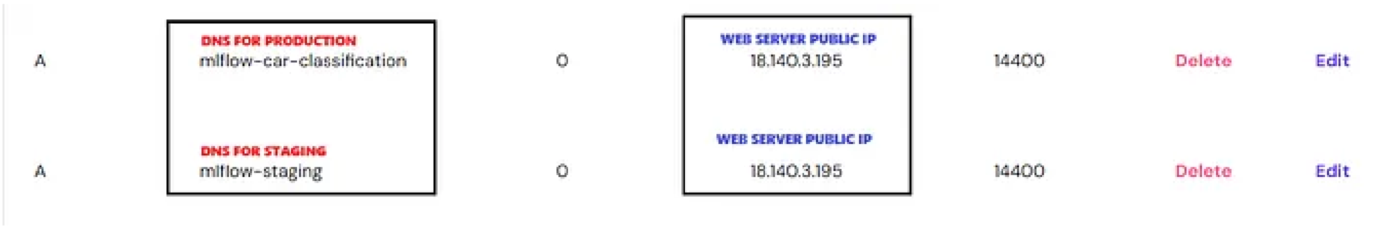
The web server is responsible for managing traffic and ensuring secure access to your application. Follow these steps to set it up:

—

Step 1: Create DNS (Optional)

Since this is a personal project:

- I created two DNS entries: one for the **staging server** and another for the **production server**.
- In the DNS setup, **point the web server's public IP address to distribute traffic to the web server**. The web server will then forward traffic to the servers defined in the NGINX configuration.



A	DNS FOR PRODUCTION mlflow-car-classification	0	WEB SERVER PUBLIC IP 18.140.3.195	14400	Delete	Edit
A	DNS FOR STAGING mlflow-staging	0	WEB SERVER PUBLIC IP 18.140.3.195	14400	Delete	Edit

DNS Configuration

> *Note:* *In a real-world scenario, a company typically already has a DNS configured by the DevOps or IT team. You would simply use the existing DNS and configure it in the web server.*

—

Step 2: Connect to Your Web Server

Connect to web server using SSH.

—

Step 3: Install NGINX and Certbot

- Update your package list and install NGINX and Certbot.

```
sudo apt update
sudo apt install nginx certbot python3-certbot-nginx
```

- Start and enable NGINX.

```
sudo systemctl start nginx
sudo systemctl enable nginx
```

—

Step 4: Configure Load Balancer and Reverse Proxy

- Navigate to the NGINX configuration directory.

```
cd /etc/nginx/sites-available
```

- Create a new configuration file for your site.

```
sudo nano your_staging_or_production_domain.conf
```

- Add load balancer and reverse proxy configuration.

```
upstream <your_upstream_name> {
    server <your_server_public_IP>:<selected_port>;
    server <your_server_public_IP>:<selected_port>;
    # Add more servers if needed
}
server {
    listen 80; # Default HTTP port
    server_name <your_domain_name>; # Replace with your domain or public IP

    location / {
        proxy_pass http://<your_upstream_name>; # Redirect requests to the load balancer
    }
}
```

1. Replace `<your_upstream_name>` with a name for your server group.
2. Replace `<your_server_public_IP>` with your staging/production server's public IP.

3. Replace `<selected_port>` with the port your app is running on.
4. Replace `<your_domain_name>` with your staging/production domain or public IP.
5. Replace `<your_upstream_name>` with the name you used in the `upstream` block.

- Example configuration:

```
# Load Balancer Config
upstream my_production_servers {
    server 192.168.1.10:5000;
    server 192.168.1.11:5000;
    server 192.168.1.12:5000;
}

# Reverse Proxy Config
server {
    listen 80;
    server_name mlflow-production.rcofwork.cloud;

    location / {
        proxy_pass http://my_production_servers;
    }
}
```

Since I deployed my application to both **staging** and **production** servers, I created separate NGINX configurations for each environment. The process is the same for both. The only difference is replacing details like **server IPs**, **domain names**, and **ports** with the specific information for staging or production.

- Enable the configuration

```
sudo ln -s /etc/nginx/sites-available/your_staging/production_domain.conf /etc/nginx/sites-enabled/
```

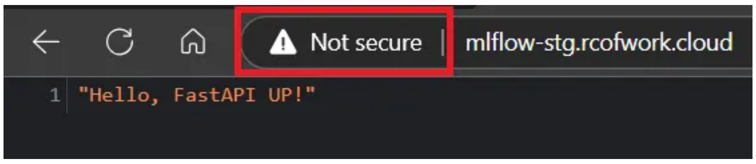
- Test the NGINX configuration

```
sudo nginx -t
```

- Restart NGINX to apply the changes:

```
sudo systemctl restart nginx
```

- Verify the setup by visiting your domain in a web browser. At this point, the site will work but will not yet be secure (HTTP only).



```
"Hello, FastAPI UP!"
```

Not Secure App

—

Step 5: Create an SSL Certificate

Use Certbot to obtain and install an SSL certificate.

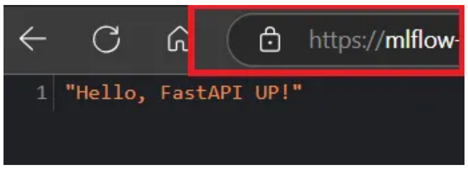
```
sudo certbot --nginx -d "www.your_staging_or_production_domain.com"
```

Certbot will update your NGINX configuration to use HTTPS and redirect HTTP traffic to HTTPS.

—

Step 6: Restart the App and Verify SSL

- Restart your app (e.g., Docker Compose).
- Verify the SSL setup by visiting your domain in a web browser. You should see a padlock icon, meaning the connection is secure



```
"Hello, FastAPI UP!"
```

Secure App

. . .

Set Up Local Environment

Follow these steps to develop the machine learning workflow in your local environment:

—

Step 1: Set up virtual environment

- Run the following command to create a virtual environment

```
python -m venv your_project_name
```

- Activate the virtual environment

```
# For Linux/WSL
source your_project_name/bin/activate

# For Windows
your_project_name\Scripts\activate
```

—

Step 2: Set Up Project Directories

- Organize your project by creating a clear folder structure.

—

Step 3: Install Required Packages

- Install the necessary Python packages for this project.

Check out **my repository** to see the project template and the Python packages I used.

. . .

Set Up Local Repositories

In this project, I used **Git** and **DVC** to manage the local repositories:

- **Git:** Stores and tracks changes in the source code.
- **DVC (Data Version Control):** Stores and tracks changes in the model dataset.

While Git handles code versioning, DVC is designed for managing large files like machine learning datasets. **DVC works alongside Git but serves a different purpose, which is why DVC must be initialized after Git.**

—

Step 1: Set Up Git

- In your project's root directory, initialize a Git repository by running

```
git init
```

—

Step 2: Set Up DVC

- After setting up Git, initialize DVC by running

```
dvc init
```

. . .

Set Up Online Repositories

After initializing your local repositories, the next step is to connect them to online repositories. In this project, I used:

- **GitHub** for source code management
- **MinIO** for model dataset storage.

—

Step 1: Connect Git to GitHub

This repository will store your source code and enable CI/CD with GitHub Actions.

Set Up GitHub Repository

- If you're new to GitHub, check **this article** to set up your repository using SSH.
- If you're already familiar, go ahead and create a repository.
- Once your repository is created, connect your local Git repository to GitHub by running:

```
git remote add origin ssh_repository_url
```

- Replace `ssh_repository_url` with the actual SSH URL of your GitHub repository.

—

Step 2: Connect DVC to MinIO

To track datasets using DVC and store them in MinIO, follow these steps:

Add DVC default remote

- Point DVC to the MinIO bucket where your datasets will be stored.

```
dvc remote add -d minio s3://your_MinIO_bucket_name
```

Set MinIO endpoint URL

- Configure DVC to use your MinIO server by specifying its endpoint URL.

```
dvc remote modify minio endpointurl https://<MinIO_service_public_ip>:<MinIO_port>
```

Set MinIO Credentials

- Access the **MinIO web UI.**
- Navigate to **Access Key Menu** → **Create Access and Secret Key**.
- **Copy and save** the keys, as they are shown only once.

Now, configure DVC with these credentials:

```
# Add MinIO access key
dvc remote modify minio access_key <your_minio_access_key>

# Add MinIO secret access key
dvc remote modify minio secret_access_key <your_minio_secret_access_key>
```

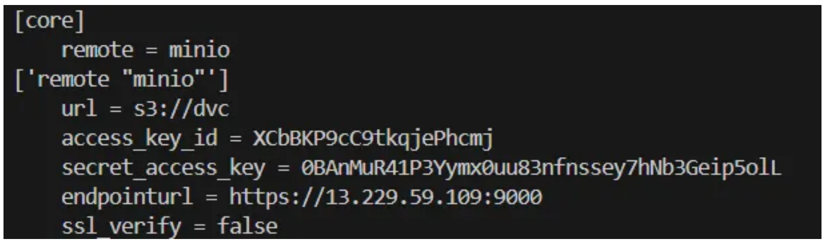
Disable SSL Verification (Optional)

- If you're using a self-signed certificate, disable SSL verification:

```
dvc remote modify minio ssl_verify false
```

Verify the Config

- If you run **`cat .dvc/config`**, the output will look like this:



```
[core]
    remote = minio
['remote "minio"']
    url = s3://dvc
    access_key_id = XCbBKP9cC9tkqjePhcmj
    secret_access_key = 0BAnMuR41P3Yymx0uu83nfnssey7hNb3Geip5olL
    endpointurl = https://13.229.59.109:9000
    ssl_verify = false
```

DVC Config

. . .

Track and Push Dataset Changes

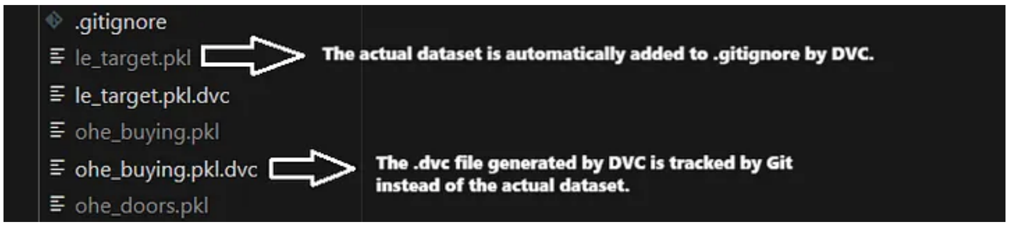
After initializing and connecting your local and remote repositories, you can now verify the setup by pushing data, files, and code changes while tracking them properly.

—

Step 1: Add dataset to DVC

```
dvc add path/to/dataset/you/want/to/add
```

- DVC will automatically create a `.gitignore` file for the dataset and **generate a `.dvc` file that Git will track instead of the actual dataset.**



```
.gitignore
le_target.pkl
le_target.pkl.dvc
ohe_buying.pkl
ohe_buying.pkl.dvc
ohe_doors.pkl
```

The actual dataset is automatically added to .gitignore by DVC.

The .dvc file generated by DVC is tracked by Git instead of the actual dataset.

DVC Add Result

—

Step 2: Commit DVC changes

- Commit the changes made by DVC.

```
dvc commit
```

—

Step 3: Add and Commit changes in Git

- Add the `.dvc` file and other necessary changes to Git, then commit them:

```
# Add changes to Git
git add your/selected/files

# Commit Git changes
git commit -m "commit message"
```

—

Step 4: Push DVC Changes to MinIO

- Upload your dataset to MinIO

```
dvc push
```

—

Step 5: Push Git Changes to GitHub

- Finally, push your Git changes to GitHub:

```
git push origin main
```

. . .

Experimenting, Evaluating and Comparing Models

Now that the local project environment is set up, you can begin developing machine learning models and integrating them with MLflow to manage the machine learning lifecycle while tracking changes with Git and DVC

—

Step 1: Exploratory Data Analysis (EDA)

I started with Exploratory Data Analysis (EDA) to better understand the dataset. This step helps in:

- Identifying the characteristics of the data.
- Finding potential improvements or additional features.

—

Step 2: Data Preprocessing

To prepare the data for training, I applied:

- **Label Creation**: Assigning target variables.
- **One-Hot Encoding**: Converting categorical variables into numerical format.

—

Step 3: Model Training

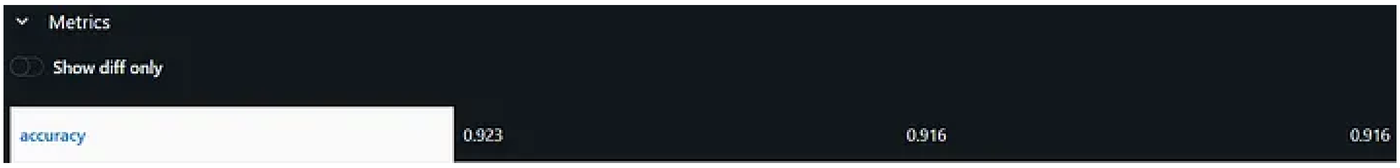
I trained the model under three different conditions:

- **Untouched Model:** Default settings without modifications.
- **Chosen Settings:** Configurations tailored to improve performance.
- **Hyperparameter Tuning:** Optimizing parameters for the best results.

During training, I integrated MLflow to log:

- **Metrics** (e.g., accuracy, loss, F1-score).
- **Parameters** (e.g., learning rate, batch size).
- **Artifacts** (e.g., trained models, plots).

These logs can be viewed in the **MLflow Web UI** for analysis.



accuracy	0.923	0.916	0.916

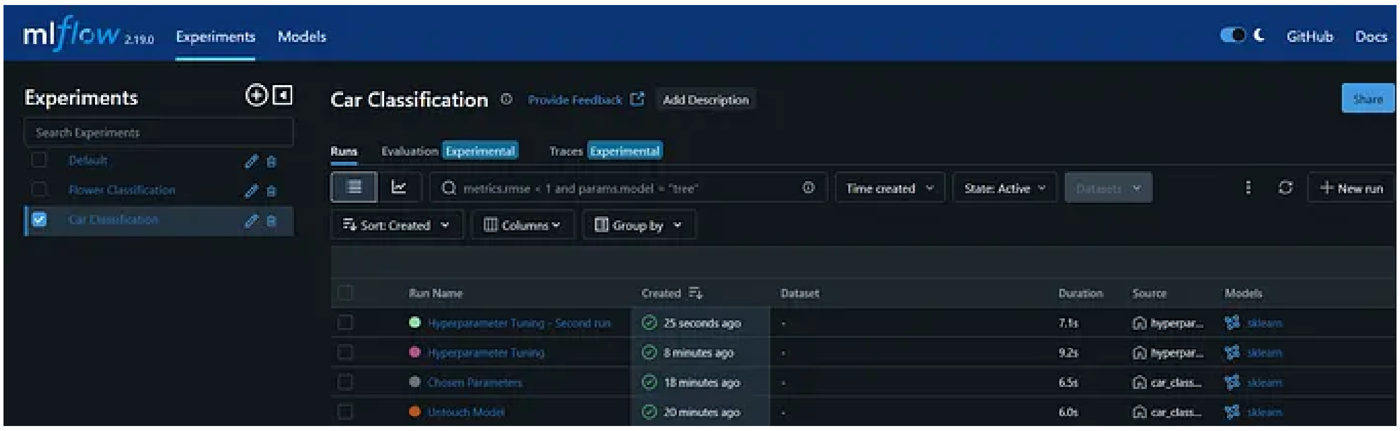
MLflow Log Accuracy Example

—

Step 4: Evaluate and Compare Models

Once the models were trained, I used the **MLflow Web UI** to:

- Compare the performance of different models.
- Analyze metrics and logs to determine the best-performing model.

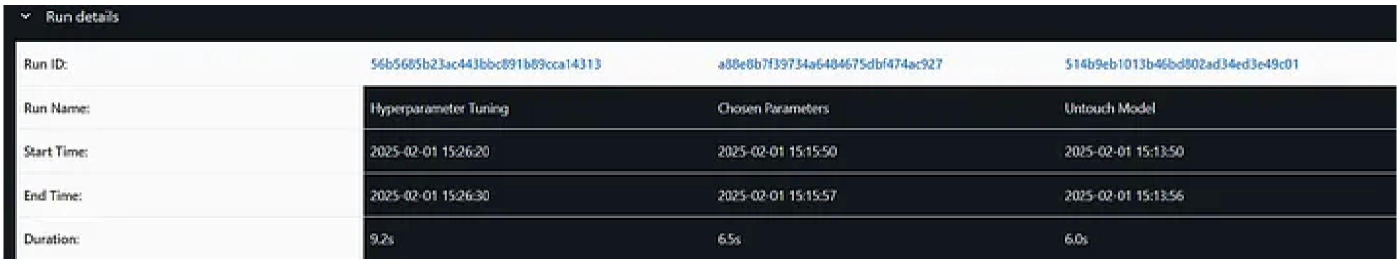


	Run Name	Created	Dataset	Duration	Source	Models
☐	● Hyperparameter Tuning - Second run	✓ 25 seconds ago	-	7.1s	hyperpar...	sklearn
☐	● Hyperparameter Tuning	✓ 8 minutes ago	-	9.2s	hyperpar...	sklearn
☐	● Chosen Parameters	✓ 18 minutes ago	-	6.5s	car_class...	sklearn
☐	● Untouch Model	✓ 20 minutes ago	-	6.0s	car_class...	sklearn

MLflow Web UI

The **MLflow Web UI** helps:

- Visualize experiment results.
- Track model improvements over time.
- Select the optimal model for deployment.



Run ID:	56b5685b23ac443bbc891b89cca14313	a88e8b7f39734a6484675dbf474ac927	514b9eb1013b46bd802ad34ed3e49c01
Run Name:	Hyperparameter Tuning	Chosen Parameters	Untouch Model
Start Time:	2025-02-01 15:26:20	2025-02-01 15:15:50	2025-02-01 15:13:50
End Time:	2025-02-01 15:26:30	2025-02-01 15:15:57	2025-02-01 15:13:56
Duration:	9.2s	6.5s	6.0s

Comparing Models in MLflow — 1

Check out **my repository** to see how I developed the machine learning workflow and integrated it with MLflow.

. . .

Unit Test in Local

After determining the best model, the next step is to test it locally. This helps catch bugs and identify areas for improvement before deployment.

—

Step 1: Create an API

In this project, I built a simple API to test the model. If you'd like to build a frontend later, that's optional and up to you.

The API does two things:

- Loads the selected model from MLflow.
- Generates predictions based on the input data.

This allows interaction with the model to verify its functionality

—

Step 2: Create a Unit Test

To ensure the model works as expected, I wrote basic unit tests focusing on:

- **Model Availability:** Verifies that the model loads successfully from MLflow.
- **Prediction Test:** Checks if the model can generate predictions for sample input.

The objective of these tests is to confirm that:

- The model can be accessed without errors.
- The model is capable of running predictions as expected.

These tests will also be used later in the CI/CD pipeline to ensure the model performs consistently across different environments.

. . .

Automating Deployment with CI/CD Using GitHub Actions

After running unit tests successfully, I set up a **CI/CD workflow** to automate the deployment process. Here's how I did it:

—

Step 1: Create a Dockerfile

I created a Dockerfile **to package the API into a container image**. This image serves as the foundation for deploying the application using Docker Compose.

- Check out the Dockerfile configuration **here**.

—

Step 2: Create Docker Compose

Next, I created a Docker Compose file to define the required services for deployment:

- **mlflow_api:** Runs the API in a testing environment during the CI (Continuous Integration) stage.
- **api_stg:** Deploys the application to the staging server for further testing.
- **api_prod:** Deploys the application to the production server for end users.

Check out the full Docker Compose configuration **here**.

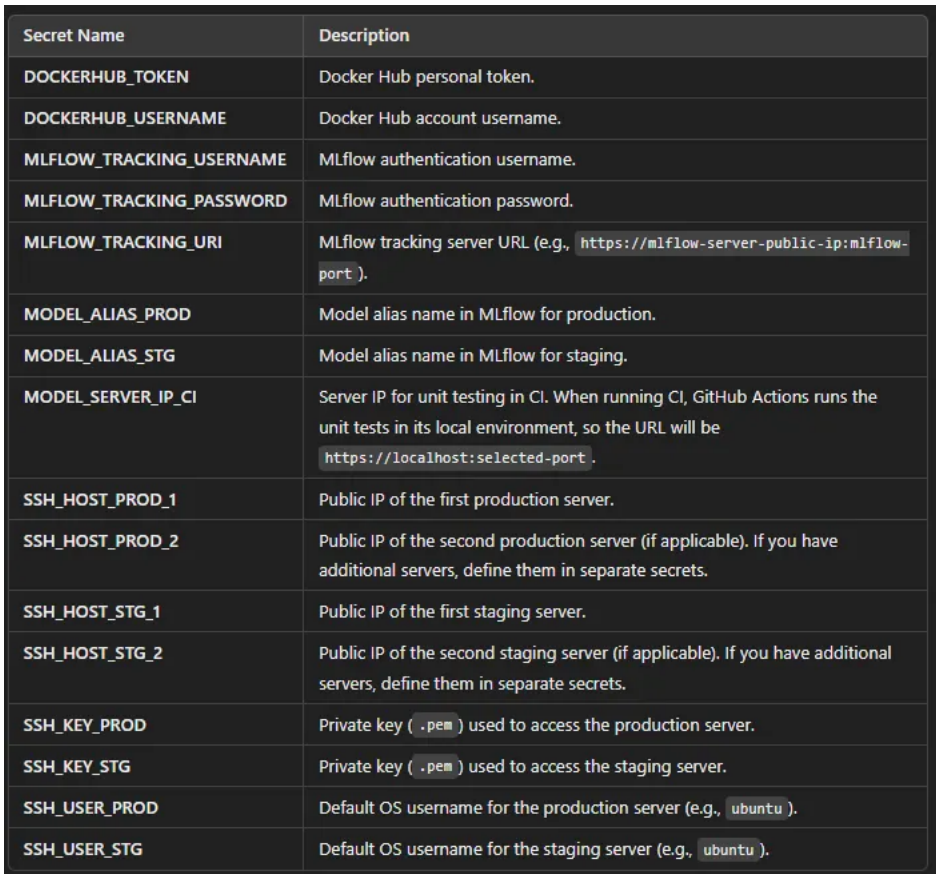
—

Step 3: Set Up CI/CD Workflow with GitHub Actions

To automate deployment, I used **GitHub Actions** to create a CI/CD pipeline. Here's how it works:

Create GitHub Secrets

- Before setting up the workflow scripts, I stored sensitive information (such as credentials and API keys) in **GitHub Secrets**. Thes are securely accessed during the CI/CD process. Here are the details:



Secret Name	Description
DOCKERHUB_TOKEN	Docker Hub personal token.
DOCKERHUB_USERNAME	Docker Hub account username.
MLFLOW_TRACKING_USERNAME	MLflow authentication username.
MLFLOW_TRACKING_PASSWORD	MLflow authentication password.
MLFLOW_TRACKING_URI	MLflow tracking server URL (e.g., `https://mlflow-server-public-ip:mlflow-port`).
MODEL_ALIAS_PROD	Model alias name in MLflow for production.
MODEL_ALIAS_STG	Model alias name in MLflow for staging.
MODEL_SERVER_IP_CI	Server IP for unit testing in CI. When running CI, GitHub Actions runs the unit tests in its local environment, so the URL will be `https://localhost:selected-port`.
SSH_HOST_PROD_1	Public IP of the first production server.
SSH_HOST_PROD_2	Public IP of the second production server (if applicable). If you have additional servers, define them in separate secrets.
SSH_HOST_STG_1	Public IP of the first staging server.
SSH_HOST_STG_2	Public IP of the second staging server (if applicable). If you have additional servers, define them in separate secrets.
SSH_KEY_PROD	Private key (`.pem`) used to access the production server.
SSH_KEY_STG	Private key (`.pem`) used to access the staging server.
SSH_USER_PROD	Default OS username for the production server (e.g., `ubuntu`).
SSH_USER_STG	Default OS username for the staging server (e.g., `ubuntu`).

GitHub Secrets Description

—

Set Up Continuous Integration (CI) Workflow

Trigger:
This workflow runs when there is a **pull request to the main branch.** If the tests pass, the code can be merged into the main branch.

What Happens in This Workflow:

- Builds the `mlflow_api` service container using the `docker-compose.yaml` file.
- Installs all required dependencies and packages.
- Runs unit tests to ensure the application works as expected.

—

Deploy to Staging Server (CD Workflow)

Trigger:
This workflow runs on a **push to the main branch.** After the CI tests pass, the app is automatically deployed to staging.

What Happens in This Workflow:

- Builds the `api_stg` service container and pushes it to a container registry (e.g., Docker Hub).
- Creates an SSH configuration file to connect to the staging server.
- Creates a `.env` file to store environment variables.
- Deploys the application to the selected staging server.

Verify the Results:
Open the staging domain in your web browser to confirm the app is running sucessfuly.

—

Performed Load Testing

After successfully deploying the application to the staging server, I performed **load testing** to evaluate its performance under stress. This helps identify potential issues before moving to production.

I used **Locust** to simulate multiple users accessing the application simultaneously. Here's how I did it:

Create a Locust Test Script

The test script performs the following tasks:

- Sends a `GET` request to the `/` endpoint.
- Sends a request to the `/predict` endpoint.
- Generates random data to fill the API input.
- Sends a `POST` request to the `/predict` endpoint.

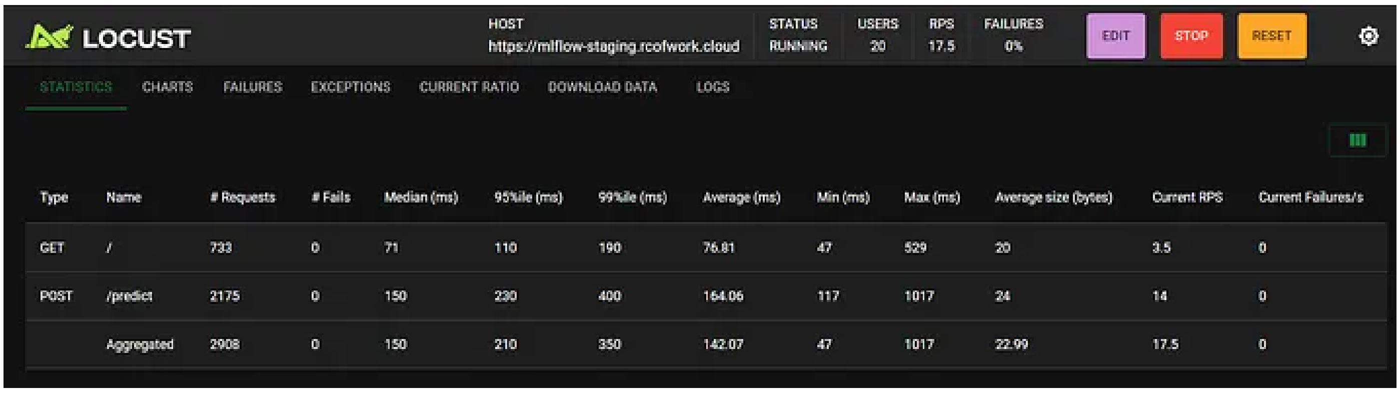
Check out the test script **here**.

Run Locust

```
locust -f locustfile.py
```

Analyze the Results

- Monitor key performance metrics such as **response time**, **requests per second** and **error rates**.
- Identify slow endpoints or high failure rates and optimize the application accordingly.



HOST https://mlflow-staging.rcofwork.cloud | STATUS RUNNING | USERS 20 | RPS 17.5 | FAILURES 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	733	0	71	110	190	76.81	47	529	20	3.5	0
POST	/predict	2175	0	150	230	400	164.06	117	1017	24	14	0
	Aggregated	2908	0	150	210	350	142.07	47	1017	22.99	17.5	0

Load Testing using Locust

Once the application passed load testing and performed well under stress, it was ready for **production deployment**.

—

Deploy to Production Server (CD Workflow)

Trigger:
This workflow runs when a **release is published or edited** in GitHub. Once ready, the application is deployed for end users.

What Happens in This Workflow:

- Similar to the staging deployment, but with one key difference:
- When building and pushing the container image, a **version tag** is added to indicate the production version.

Verify the Results:
Go to the production domain in your web browser to confirm the app is running successfully. This is the domain that end users will visit

Check out **my repository** to see my CI/CD worfklow and more!

. . .

Key Takeaways

That's the final step in this project! You've reached the end of this article. To summarize, I shared my exploration of MLOps, including:

- **Setting up cloud infrastructure.**
- **Using MLflow for experiment tracking.**
- **Configuring a web server.**
- **Performing unit tests and load testing.**
- **Automating deployment with CI/CD.**
- **Ensuring secure access for end users.**

Thank you for following along on this learning journey. I hope you gained useful insights to apply to your own projects!

To explore the full source code for this project, check out my repository **here**.

. . .

If you have any questions, please feel free to contact me. I welcome any feedback or suggestions you may have.

You can connect with me on:

- My LinkedIn
- My GitHub

Machine Learning Mlops Mlflow Data Engineering Data Science

Written by Rico Febrian

35 followers · 3 following

Hi, Welcome to My Learning Logs!

Responses (2)

Write a response

What are your thoughts?

Rico Febrian Author
Jun 2

You can check out the full dataset here.

Thank you for highlighting this, I just fixed the dataset link!

Reply

Ramsha javed
May 24

good article

1 reply Reply

More from Rico Febrian

See all from Rico Febrian

Recommended from Medium

See more recommendations