



Building an ML Platform from Scratch

How to Build an ML Platform with Open Source Tools

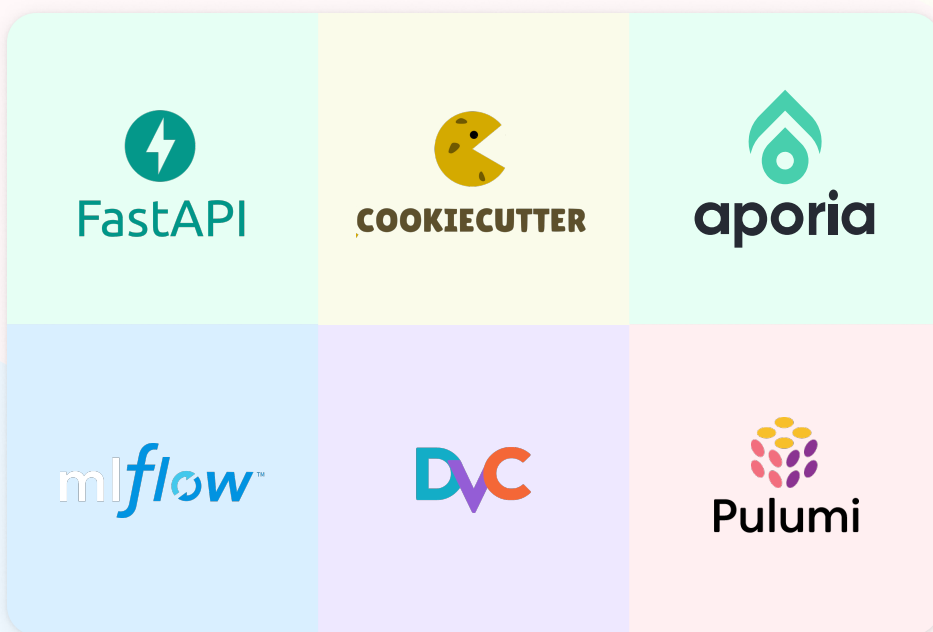


Table of Contents

| | |
|---|----|
| Introduction | 1 |
| Architecture | 2 |
| Let's start building! | 5 |
| MLFlow Installation | 6 |
| Exposing K8s apps to the Internet using Traefik | 7 |
| Model Template: Training | 12 |
| Model Template: Serving | 16 |
| Model Template: Monitoring | 21 |
| Model Template: CI/CD | 26 |
| Model Template: DVC Integration | 29 |
| Model Template: Cookiecutter | 31 |
| What's missing from this? | 32 |
| Summary | 23 |

Introduction

As your data science team grows and you start deploying models to production, the need for proper ML infrastructure becomes crucial – a standard way to design, train and deploy models.

In this guide, together we will build a basic ML Platform using open-source tools like Cookiecutter, DVC, MLFlow, FastAPI, Pulumi, and more. We'll also see how to monitor for model drift using [Aporia's](#) Cloud-Native ML Observability solution. [Final code is available on GitHub](#).

Keep in mind that this type of project can be huge – often taking a lot of time and resources – therefore our toy ML Platform won't have tons of features – just the basics, but it should teach you the basic principles of how to build your own ML platform.

Architecture

Our toy ML Platform will use [DVC](#) for data versioning, [MLFlow](#) for experiments management, [FastAPI](#) for model serving, and [Aporia](#) for model monitoring.

We're going to build all of this on top of AWS, but in theory, you could also use Azure, Google Cloud, or any other cloud provider.

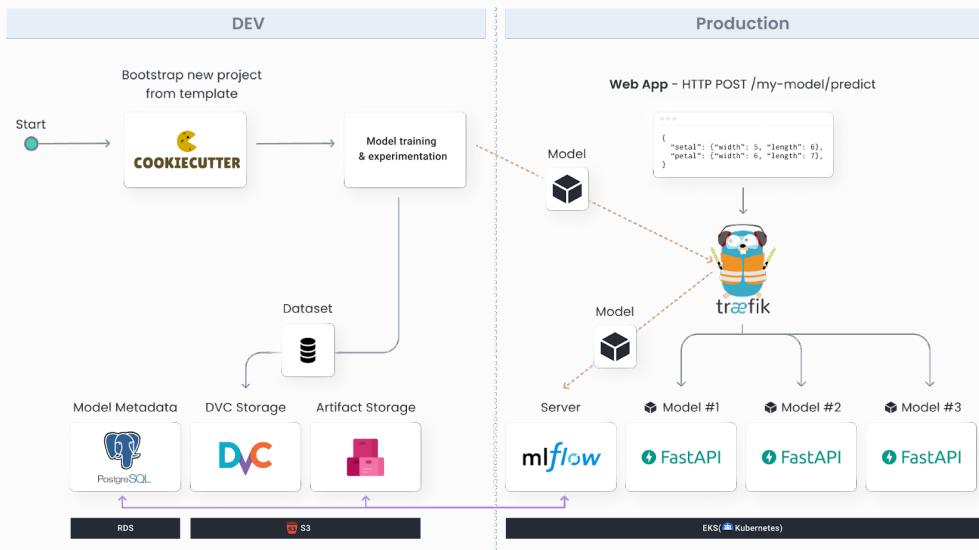
It's important to note that when building your own machine learning platform, you should NOT take these tools for granted. You should [evaluate the alternatives](#) – as they may be more appropriate for your specific use case.

Model Template

The first component in our machine learning platform is going to be the model template, which we're going to build using Cookiecutter for templating, and Poetry for package management.

The idea is that when a data scientist starts working on a new project, they will clone our model template (which contains a standard folder structure, Python linting, etc), develop their model, and easily deploy it when it's ready for production.

The machine learning model template will contain basic training and serving code.



Building an ML Platform Model Template

Data & Experiment Tracking

The training code in the model template will use the MLFlow client to track experiments.

Those experiments will be sent to the MLFlow Server that we'll run on top of Kubernetes (EKS).

The model artifact itself is going to be saved in an S3 Bucket (the Artifact Storage), and metadata about experiments will be saved in a PostgreSQL database.

We'll also track versions of the dataset using DVC, in an S3 bucket.

Model Serving

For model serving, we will build a FastAPI server that'll be responsible for preprocessing, making predictions, etc.

These model servers are going to run on Kubernetes, and we'll expose them to the internet using [Traefik](#).

Infrastructure as Code

All our infrastructure is going to be deployed using [Pulumi](#), which is an Infrastructure as Code tool similar to Terraform.

If you aren't familiar with the concept, [you can read more about it here](#) before continuing. Here are some major advantages of using this method:

- **Versioned:** Your infrastructure is versioned, so if you have a bug you can easily revert it to a previous version.
- **Code Reviewed:** Each change to the infrastructure can be code reviewed and you're less prone to mistakes.
- **Sharable:** You can easily share infrastructure components, just send the source code for the component.

With Pulumi, you can choose to write your infrastructure in a real programming language, such as TypeScript, Python, C#, and more.

Even though the natural choice for an ML platform would be Python, I chose TypeScript because, at the time of writing this post (July 2021), Pulumi's implementation of TypeScript is more feature complete.

Repositories & CI/CD

We're going to have 2 GitHub repositories:

- **mlplatform-infra** – the Pulumi code for the shared infrastructure of the ML Platform. Infrastructure that isn't model-specific. Kubernetes, MLFlow, S3 buckets, etc.
- **model-template** – the model template code that data scientists can clone, including basic training code, FastAPI server, etc.

For CI/CD we're going to use [GitHub Actions](#).

Let's start building!

Let's start by setting up Kubernetes with an MLFlow server.

Create a new GitHub repo called mlplatform-infra. This repo will contain the underlying infrastructure that's shared between models.

Clone it and run:

```
pulumi new
```

Open index.ts and write the code for creating a new EKS cluster:

```
1 // Create a Kubernetes cluster.
2 const cluster = new eks.Cluster('mlplatform-eks', {
3   createOidcProvider: true,
4 });
5
6 export const kubeconfig = cluster.kubeconfig;
```

The createOidcProvider is required because MLFlow is going to access the artifact storage (see architecture), which is a S3 bucket, so we need to create a Kubernetes ServiceAccount that can access S3 buckets.

[There are more interesting arguments in this API](#), you should definitely check them out before deploying to production.

MLFlow Installation

To install packages like MLFlow on Kubernetes, we're going to use [Helm](#), which is a very popular package manager for Kubernetes.

Specifically, we're going to use the [labirras/mlflow](#) Helm chart:

```
1 // Install MLFlow
2 const mlflow = new k8s.helm.v3.Chart("mlflow", {
3   chart: "mlflow",
4   fetchOpts: { repo: "https://larribas.me/helm-charts" },
5 }, { provider: cluster.provider });
```

This should install an MLFlow server, but unfortunately it's not going to work because we still need to connect it to an artifact storage and a model metadata database.

Let's create an S3 bucket for the artifact storage, and configure MLFlow to use it using the Helm chart's values:

```
1 // Create S3 bucket for MLFlow artifact storage
2 const artifactStorage = new aws.s3.Bucket("artifact-
3   storage", {
4   acl: "public-read-write",
5 });
6 // Install MLFlow
7 const mlflow = new k8s.helm.v3.Chart("mlflow", {
8   chart: "mlflow",
9   values: {
10     defaultArtifactRoot:
11       artifactStorage.bucket.apply((bucketName: string) =>
12         `s3://${bucketName}`),
13   },
```

```
12   fetchOpts: { repo: "https://larribas.me/helm-  
charts" },  
13 }, { provider: cluster.provider });
```

This should almost work, but there's one problem: the MLFlow server, which is running on Kubernetes, isn't going to have access to S3 buckets (see comment above on `createOidcProvider`).

To fix the permissions issue, let's create a `ServiceAccount` that can access S3 buckets and use it:

```
1 // Create S3 bucket for MLFlow artifact storage  
2 const artifactStorage = new aws.s3.Bucket("artifact-  
storage", {  
3   acl: "public-read-write",  
4 });  
5  
6 // Create a k8s ServiceAccount that can access S3  
  buckets for MLFlow  
7 const mlflowServiceAccount = new  
  S3ServiceAccount('mlflow-service-account', {  
8   namespace: "default",  
9   oidcProvider: cluster.core.oidcProvider!,  
10  readOnly: false,  
11 }, { provider: cluster.provider });  
12  
13 // Install MLFlow  
14 const mlflow = new k8s.helm.v3.Chart("mlflow", {  
15   chart: "mlflow",  
16   values: {  
17     defaultArtifactRoot:  
      artifactStorage.bucket.apply((bucketName: string) =>  
        `s3://${bucketName}`),  
18     serviceAccount: {  
19       create: false,
```

```
20     name: mlflowServiceAccount.name,  
21   },  
22 },  
23   fetchOpts: { repo: "https://larribas.me/helm-  
charts" },  
24 }, { provider: cluster.provider });
```

The S3ServiceAccount is available [here](#).

Finally, we need to set up the Postgres database for model metadata. We'll use RDS for that:

```
1  // Create Postgres database for MLFlow  
2  const dbPassword = new  
    random.RandomPassword('mlplatform-db-password', {  
    length: 16, special: false });  
3  const db = new aws.rds.Instance('mlflow-db', {  
4    allocatedStorage: 10,  
5    engine: "postgres",  
6    engineVersion: "11.10",  
7    instanceClass: "db.t3.medium",  
8    name: "mlflow",  
9    password: dbPassword.result,  
10   skipFinalSnapshot: true,  
11   username: "postgres",  
12  
13   // Make sure EKS has access to this db  
14   vpcSecurityGroupIds:  
    [cluster.clusterSecurityGroup.id,  
    cluster.nodeSecurityGroup.id],  
15  });  
16  
17  // Install MLFlow  
18  const mlflow = new k8s.helm.v3.Chart("mlflow", {  
19    chart: "mlflow",
```



```
20   values: {  
21     ...  
22     backendStore: {  
23       postgres: {  
24         username: db.username,  
25         password: db.password,  
26         host: db.address,  
27         port: db.port,  
28         database: "mlflow"  
29       }  
30     },  
31   },  
32   fetchOpts: { repo: "https://larribas.me/helm-  
charts" },  
33 }, { provider: cluster.provider });
```

That's it, run `pulumi up` and you should see MLFlow running on your new Kubernetes cluster!

Exposing K8s apps to the Internet using Traefik

Even though MLFlow is running successfully, you have no way of accessing it. Let's open MLFlow to the internet!

Security Warning

In real life you shouldn't just expose MLFlow to the Internet of course 😊
Instead, think of an authentication & authorization mechanisms that work for your organization.

To expose Kubernetes applications to the Internet, we are going to use Traefik – an open-source API gateway.

To install Traefik:

```
1 // Install Traefik
2 const traefik = new k8s.helm.v3.Chart('traefik', {
3   chart: 'traefik',
4   fetchOpts: { repo: 'https://containous.github.io/traefik-helm-chart' },
5 }, { provider: cluster.provider })
```

To expose MLFlow on the Traefik we just installed:

```
1 // Expose MLFlow in Traefik as /mlflow
2 new TraefikRoute('mlflow', {
3   prefix: '/mlflow',
4   service: mlflow.getResource('v1/Service', 'mlflow',
5   'mlflow'),
6   namespace: 'default',
7 }, { provider: cluster.provider, dependsOn: [mlflow]
8   });
```

The TraefikRoute component is available [here](#).

To get Traefik's public hostname, add the following line:

```
1 export const hostname = traefik.getResource('v1/
  Service',
  'traefik').status.loadBalancer.ingress[0].hostname;
```

After running `pulumi up` again, you should be able to get that hostname by running:

```
pulumi stack output hostname
```

You can now create a CNAME record in your domain's DNS provider. You should now be able to access MLFlow at <http://yourdomain.com/mlflow> 😊

Security Warning

In real-life you would want to set up HTTPS and remove HTTP in Traefik's Helm chart values. In this guide we use HTTP only.

Model Template: Training

Let's start building our model template. Clone the model-template repo and start a new Poetry package:

```
poetry new --src my_model
```

We'll call it `my_model` for now, and change to Cookiecutter variables later on.

Create a `my_model.training` package and add your training starting point. Here we'll use a simple LightGBM example:

```
1  from sklearn.model_selection import train_test_split
2  from sklearn.metrics import accuracy_score, log_loss
3  import pandas as pd
4  import lightgbm as lgb
5
6
7  # Prepare training data
8  df = pd.read_csv('data/iris.csv')
9  flower_names = {'Setosa': 0, 'Versicolor': 1,
10                  'Virginica': 2}
11
12  X = df[['sepal.length', 'sepal.width',
13          'petal.length', 'petal.width']]
14  y = df['variety'].map(flower_names)
15  X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.2,
                    random_state=42)
```

```
16
17 train_data = lgb.Dataset(X_train, label=y_train)
18
19 def main():
20     # Train model
21     params = {
22         "objective": "multiclass",
23         "num_class": 3,
24         "learning_rate": 0.2,
25         "metric": "multi_logloss",
26         "feature_fraction": 0.8,
27         "bagging_fraction": 0.9,
28         "seed": 42,
29     }
30
31     model = lgb.train(params, train_data,
32                       valid_sets=[train_data])
33
34     # Evaluate model
35     y_proba = model.predict(X_test)
36     y_pred = y_proba.argmax(axis=1)
37
38     loss = log_loss(y_test, y_proba)
39     acc = accuracy_score(y_test, y_pred)
40
41 if __name__ == "__main__":
42     main()
```

To make it work, you will need to iris.csv dataset from [here](#).

You can also add a Poetry script to make training easy to run. Add this to your pyproject.toml file:

```
1 [tool.poetry.scripts]
2 train = "src.my_model.training.train:main"
```

And now you can run:

```
poetry run train
```

OK, let's add MLFlow client to this training code. I've highlighted the changes:

```
1  from sklearn.model_selection import train_test_split
2  from sklearn.metrics import accuracy_score, log_loss
3  import pandas as pd
4  import lightgbm as lgb
5  import mlflow
6  import mlflow.lightgbm
7
8
9  # Enable auto logging
10 mlflow.set_tracking_uri('http://yourdomain.com/
    mlflow')
11 mlflow.lightgbm.autolog()
12
13
14 # Prepare training data
15 df = pd.read_csv('data/iris.csv')
16 flower_names = {'Setosa': 0, 'Versicolor': 1,
    'Virginica': 2}
17
18
19 X = df[['sepal.length', 'sepal.width',
    'petal.length', 'petal.width']]
20 y = df['variety'].map(flower_names)
21
22 X_train, X_test, y_train, y_test =
23 train_test_split(X, y, test_size=0.2,
    random_state=42)
24
25 train_data = lgb.Dataset(X_train, label=y_train)
```

```
25
26 def main():
27     with mlflow.start_run() as run:
28         # Train model
29         params = {
30             "objective": "multiclass",
31             "num_class": 3,
32             "learning_rate": 0.2,
33             "metric": "multi_logloss",
34             "feature_fraction": 0.8,
35             "bagging_fraction": 0.9,
36             "seed": 42,
37         }
38
39         model = lgb.train(params, train_data,
40                           valid_sets=[train_data])
41
42         # Evaluate model
43         y_proba = model.predict(X_test)
44         y_pred = y_proba.argmax(axis=1)
45
46         loss = log_loss(y_test, y_proba)
47         acc = accuracy_score(y_test, y_pred)
48
49         # Log custom metrics if you want
50         mlflow.log_metrics({
51             "log_loss": loss,
52             "accuracy": acc
53         })
54
55         print("Run ID:", run.info.run_id)
56
57 if __name__ == "__main__":
58     main()
```

Try to train the model again and go to your MLFlow server. You should now see the new experiment and download the model file from the browser 😊

Model Template: Serving

Let's implement the model server based on FastAPI.

Create a **my_model.serving** package, and create a basic FastAPI server that can make predictions for your basic model. In our case:

```
1  import os
2  import uvicorn
3  import mlflow
4  import numpy as np
5  import pandas as pd
6  from fastapi import FastAPI, Request
7  from pydantic import BaseModel
8
9
10 class FlowerPartSize(BaseModel):
11     length: float
12     width: float
13
14 class PredictRequest(BaseModel):
15     sepal: FlowerPartSize
16     petal: FlowerPartSize
17
18
19 app = FastAPI()
20
21 # Load model
22 model = mlflow.lightgbm.load_model(f'runs:/
    {os.environ["MLFLOW_RUN_ID"]}/model')
23
24 flower_name_by_index = {0: 'Setosa', 1:
    'Versicolor', 2: 'Virginica'}
```



```
25
26
27 @app.post("/predict")
28 def predict(request: PredictRequest):
29     df = pd.DataFrame(columns=['sepal.length',
30                               'sepal.width', 'petal.length', 'petal.width'],
31                       data=[[request.sepal.length,
32                               request.sepal.width, request.petal.length,
33                               request.petal.width]])
34
35
36 def main():
37     uvicorn.run(app, host="0.0.0.0", port=8000)
38
39 if __name__ == "__main__":
40     main()
```

Note that the model is loaded from the Artifact Storage (the S3 bucket) through MLFlow.

OK, we'll now need to do some DevOps to make this server run on our Kubernetes. Let's start by containerizing it.

Add a Dockerfile:

```
1 FROM python:3.8-slim
2 WORKDIR /my_model
3 STOPSIGNAL SIGINT
4
5 ENV LISTEN_PORT 80
6
7 # System dependencies
8 RUN apt update && apt install -y libgomp1
```

```
9  RUN pip3 install poetry
10
11  # Project dependencies
12  COPY poetry.lock pyproject.toml ./
13
14  RUN poetry config virtualenvs.create false
15  RUN poetry install --no-interaction --no-ansi --no-dev
16
17  COPY . .
18
19  WORKDIR /my_model/src
20  ENTRYPOINT uvicorn my_model.serving.main:app --host
    0.0.0.0 --port $LISTEN_PORT --workers 2
```

Next, let's create a Pulumi package for the model that can build & push this Docker image to ECR, and deploy it to Kubernetes.

Create an infra directory inside the model template and run `pulumi new`, as before.

The Pulumi code should look something like:

```
1  import * as pulumi from '@pulumi/pulumi';
2  import * as awsx from '@pulumi/awsx';
3  import * as k8s from '@pulumi/kubernetes';
4  import * as kx from '@pulumi/kubernetesx';
5  import TraefikRoute from './TraefikRoute';
6
7  const config = new pulumi.Config();
8  const baseStack = new
    pulumi.StackReference(config.require('baseStackName'))
9
10 // Connect to the Kubernetes we created in
    mlplatform-infra
```

```
11 const provider = new k8s.Provider('provider', {
12   kubeconfig: baseStack.requireOutput('kubeconfig'),
13 })
14
15 // Build & push Docker image to ECR
16 const image = awsx.ecr.buildAndPushImage('my-model-
17   image', {
18     context: '../',
19   });
20
21 const podBuilder = new kx.PodBuilder({
22   containers: [{
23     image: image.imageValue,
24     ports: { http: 80 },
25     env: {
26       'LISTEN_PORT': '80',
27       'MLFLOW_TRACKING_URI': 'http://
28     yourcompany.com/mlflow',
29       'MLFLOW_RUN_ID': config.require('runID'),
30     }
31   }],
32   serviceAccountName:
33     baseStack.requireOutput('modelsServiceAccountName'),
34 });
35
36 const deployment = new kx.Deployment('my-model-
37   serving', {
38     spec: podBuilder.asDeploymentSpec({ replicas: 3 })
39   }, { provider });
40
41 const service = deployment.createService();
42
43 // Expose model in Traefik
44 new TraefikRoute('my-model', {
45   prefix: '/models/my-model',
46   service,
47   namespace: 'default',
48 }, { provider, dependsOn: [service] });
```

Note that on line 30 we use a special Kubernetes ServiceAccount that can read from S3 buckets.

This is similar to the ServiceAccount we created for MLFlow, but it is read-only – you can just copy-paste that piece of code in mlplatform-infra and change readOnly to true.

Model Template: Monitoring

We'll now set up a data drift monitor using [Aporia](#). For this guide we'll use Aporia's cloud. However, Aporia can also be easily installed in your private VPC.

Start by creating a free account. Then, click the "Add Model" button:

aporia

Models

All Models

My Models

Monitors

Alerts

Team

Integrations

Go live

Help

Models Management

Add model

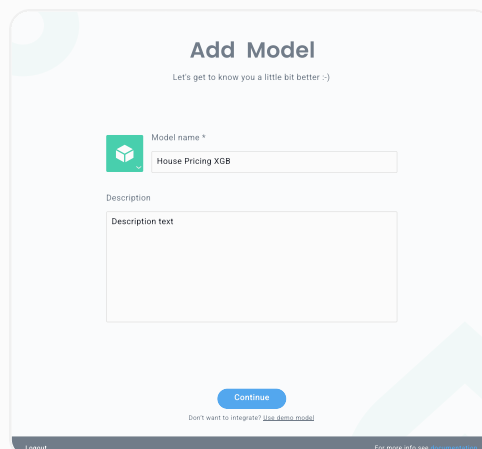
9 Models

Search

| Model Information | | Monitors | | | Model Statistics | | Integration Status | | | |
|--|-------------|----------------|---------------|-------------------|------------------|------------------------|--|--------------|--------------|--------------|
| Name | Owner | Data Integrity | Data Behavior | Model Performance | Last prediction | Avg. Predictions / Day | Activity chart | Training | Inference | Actuals |
| <div>XGB Fraud Detection</div> <div>Detects fraudulent transactions in models</div> | <div></div> | <div>✓</div> | <div>✗</div> | <div>✗</div> | a day ago | 13.08K | <div><div></div><div>13.08K</div><div>0</div><div>Days</div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> |
| <div>Suspicious Device Detection</div> | <div></div> | <div>✓</div> | <div>✓</div> | <div>✗</div> | a day ago | 3.73K | <div><div></div><div>3.73K</div><div>0</div><div>Days</div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> |
| <div>Marketing Targeting Campaign</div> | <div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> | a day ago | 108 | <div><div></div><div>108</div><div>0</div><div>Days</div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> |
| <div>LTV Model</div> | <div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> | a day ago | 1.28K | <div><div></div><div>1.28K</div><div>0</div><div>Days</div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> |
| <div>Conversion Rate</div> | <div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> | a day ago | 3.73K | <div><div></div><div>3.73K</div><div>0</div><div>Days</div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> |
| <div>Credit Risk</div> | <div></div> | <div>✓</div> | <div>✓</div> | <div>✗</div> | a day ago | 8.25K | <div><div></div><div>8.25K</div><div>0</div><div>Days</div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> |
| <div>Churn Prediction</div> | <div></div> | <div>✓</div> | <div>✗</div> | <div>✗</div> | a day ago | 1.4K | <div><div></div><div>1.4K</div><div>0</div><div>Days</div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> |
| <div>LinkedIn Sentiment Analysis</div> <div>Analyzes the sentiment of a post from LinkedIn</div> | <div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> | a day ago | 26 | <div><div></div><div>26</div><div>0</div><div>Days</div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> |
| <div>Insurance Sales Prediction</div> <div>Predict which prospects will be interested in a vehicle insurance</div> | <div></div> | <div>✓</div> | <div>✗</div> | <div>✗</div> | a day ago | 903 | <div><div></div><div>903</div><div>0</div><div>Days</div></div> | <div>✓</div> | <div>✓</div> | <div>✓</div> |

Aporia Data Point Explainer

Fill some basic details for your new model:



The screenshot shows the 'Add Model' form. It has a title 'Add Model' and a subtitle 'Let's get to know you a little bit better :-)'.

Model name *

House Pricing XGB

Description

Description text

Continue

Don't want to integrate? [Use open model](#)

Learn more

For more info see [documentation](#)

Then, integrate Aporia into your training code:

```
1  from sklearn.model_selection import train_test_split
2  from sklearn.metrics import accuracy_score, log_loss
3  import pandas as pd
4  import lightgbm as lgb
5  import mlflow
6  import mlflow.lightgbm
7  import aporia
8
9
10 aporia.init(token="<YOUR APORIA TOKEN>",
11             environment="local-dev",
12             verbose=True)
13
14
15 # ...
16
17 def main():
18     with mlflow.start_run() as run:
19         # Train model
20         # ...
21
22     # Log the new model version to Aporia
23     apr_model = aporia.create_model_version(
24         model_id="my-model",
25         model_version=run.info.run_id,
26         model_type="binary",
27         features=aporia.pandas.
28         infer_schema_from_dataframe(X),
29         predictions=aporia.pandas.
30         infer_schema_from_dataframe(y.to_frame()),
31     )
32
33     # Log training set to Aporia
34     # (only aggregations, because it can be huge)
```

```
33     apr_model.log_training_set(
34         features=X_train,
35         labels=y_train.to_frame(),
36     )
37
38     # Log test set to Aporia
39     # (only aggregations, because it can be huge)
40     apr_model.log_test_set(
41         features=X_test,
42         labels=y_test.to_frame(),
43         predictions=pd.DataFrame(data=y_pred,
44                                 columns=['variety']),
45     )
46     print("Run ID:", run.info.run_id)
47
48 if __name__ == "__main__":
49     main()
```

And integrate Aporia to your serving code:

```
1  import os
2  import uvicorn
3  import mlflow
4  import numpy as np
5  import pandas as pd
6  from fastapi import FastAPI, Request
7  from pydantic import BaseModel
8  import uuid
9  import aporia
10
11
12  aporia.init(token="<YOUR APORIA TOKEN>",
13             environment="production",
14             verbose=True)
15
```

```
16 apr_model = aporia.Model("iris-model",
17   os.environ["MLFLOW_RUN_ID"])
18 # ...
19
20
21 @app.post("/predict")
22 def predict(request: PredictRequest):
23     df = pd.DataFrame(columns=['sepal.length',
24   'sepal.width', 'petal.length', 'petal.width'],
25   data=[[request.sepal.length,
26   request.sepal.width, request.petal.length,
27   request.petal.width]])
28
29     y_pred = np.argmax(model.predict(df))
30
31     apr_model.log_prediction(
32       id=str(uuid.uuid4()),
33       features=aporia.pandas.pandas_to_dict(df),
34       predictions={"variety": y_pred},
35     )
36
37     return {"flower": flower_name_by_index[y_pred]}
38
39
40 def main():
41     uvicorn.run(app, host="0.0.0.0", port=8000)
42
43 if __name__ == "__main__":
44     main()
```


Finally, create a model drift monitor:

The screenshot shows a configuration interface for a model drift monitor. At the top, there are two input fields: "Detection:" with the text "When **Iris Model** has **Monitor Type** anomaly" and "Action:" with the text "Perform **Choose Actions**". Below these is a section titled "What would you like to monitor" with a "2/2" indicator. Inside this section, a card for "Iris Model" (Binary) is selected, indicated by a green checkmark. Below the card are two dropdown menus: "All Environments" and "All Versions". A blue "Next" button is positioned below the dropdowns. Below the "Next" button is a vertical flow diagram with three steps: "Iris Model" (selected), "Monitor Type", and "Actions". At the bottom of the flow diagram is a "Save Monitor" button.

Detection: When **Iris Model** has **Monitor Type** anomaly

Action: Perform **Choose Actions**

What would you like to monitor 2/2

Iris Model
Binary

All Environments All Versions

Next

Monitor Type

Actions

Save Monitor

Model Template: CI/CD

We'll now implement a basic CI/CD pipeline for our model template.

This pipeline will run when the data scientist pushes a commit to the main branch. It will deploy the model to our Kubernetes cluster.

We're going to use [GitHub Actions](#) for CI/CD, which is GitHub's native CI/CD system.

Our GitHub Action is going to be very simple – all it's going to do is to run make deploy, and the real deployment logic is going to be implemented in a Makefile. This is useful in case you want to deploy the model from your local computer – for example, if GitHub Actions is down and you need to deploy urgently 😊

Let's start by adding the Makefile:

```
1 SHELL := /bin/bash
2
3 STACK_NAME=dev
4 BASE_STACK_NAME=mlplatform-infra/dev
5 PULUMI_CMD=pulumi --non-interactive --stack
  $(STACK_NAME) --cwd infra/
6
7 install-deps:
8     curl -fsSL https://get.pulumi.com | sh
9     npm install -C infra/
10
11     sudo pip3 install poetry --upgrade
12     poetry install
13
14 train:
```

```
15  poetry run train
16
17  deploy:
18    $(PULUMI_CMD) stack init $(STACK_NAME) || true
19    $(PULUMI_CMD) config set aws:region $(AWS_REGION)
20    $(PULUMI_CMD) config set baseStackName
    $(BASE_STACK_NAME)
21    $(PULUMI_CMD) config set runID $(shell poetry run
    train | awk '/Run ID/{print $NF}')
22    $(PULUMI_CMD) up --yes --skip-preview
```

This Makefile contains 3 commands:

- **make install-deps**
 - Install Pulumi and its dependencies
 - Create a virtualenv and install dependencies using Poetry
- **make train**
 - Train the model
- **make deploy**
 - Train the model and take the MLFlow Run ID
 - Run Pulumi with this Run ID to deploy it to the K8s cluster

Next, we'll create the GitHub Action that deploys the model on push to the main branch.

Create `.github/workflows/deploy.yml`:

```
1  name: Deploy
2  on:
3    push:
4      branches:
5        - main
```

```
6 jobs:
7   deploy:
8     runs-on: ubuntu-20.04
9     steps:
10      - name: Checkout
11        uses: actions/checkout@master
12
13      - name: Install dependencies
14        run: make install-deps
15
16      - name: Deploy
17        run: make deploy
18        env:
19          AWS_ACCESS_KEY_ID: ${
20            secrets.AWS_ACCESS_KEY_ID }}
21          AWS_SECRET_ACCESS_KEY: ${
22            secrets.AWS_SECRET_ACCESS_KEY }}
23          AWS_REGION: ${ secrets.AWS_REGION }}
24          AWS_ECR_ACCOUNT_URL: ${
25            secrets.AWS_ECR_ACCOUNT_URL }}
26          PULUMI_ACCESS_TOKEN: ${
27            secrets.PULUMI_ACCESS_TOKEN }}
```

It takes AWS credentials and Pulumi access token from [GitHub Secrets](#).

Model Template: DVC Integration

Versioning your data is extremely important – when there's an issue with your model in production, and you need to debug it, you really want to know how your training set looked like.

If your datasets are super small you could use Git for that. If not, you'll need some other method to track versions of your datasets. In this guide, we'll use DVC, which stores your data in an S3 bucket.

First, let's create an S3 bucket for DVC. In your mlplatform-infra repo, add the following code:

```
1 // Create S3 bucket for DVC
2 const dvcBucket = new aws.s3.Bucket("dvc-bucket", {
3   acl: "public-read-write",
4 });
5
6 export const dvcBucketURI = dvcBucket.bucket.apply(
7   (bucketName: string) => `s3://${bucketName}`);
```

Next, we need to configure our model template to use this DVC repo. Inside the model template run:

```
1 cd data/
2 poetry run dvc init --subdir
3 poetry run dvc remote add --project -d s3
  <DVC_BUCKET_URI>
```

You can fetch the DVC Bucket URI from Pulumi's outputs:

```
1 pulumi stack output dvcBucketURI
```

Awesome, we should have DVC configured by now. Let's add our iris.csv to DVC:

```
1 poetry run dvc add ./iris.csv
2 poetry run dvc push
```

To make CI/CD work, we need to make sure to pull from DVC before each training. Add that to Makefile:

```
1 dvc-pull:
2   poetry run dvc --cd data/ pull
3
4 train: dvc-pull
5   poetry run train
6
7 deploy: dvc-pull
8   $(PULUMI_CMD) stack init $(STACK_NAME) || true
9   $(PULUMI_CMD) config set aws:region $(AWS_REGION)
10  $(PULUMI_CMD) config set baseStackName
    $(BASE_STACK_NAME)
11  $(PULUMI_CMD) config set runID $(shell poetry run
    train | awk '/Run ID/{print $NF}')
12  $(PULUMI_CMD) up --yes --skip-preview
```

Model Template: Cookiecutter

Cookiecutter allows you to convert our model template to a real template – when the user clones it, he'll be able to easily change the name of the model, its author, etc.

Start by adding a cookiecutter.json file:

```
1 {
2   "full_name": "Alon Gubkin",
3   "email": "alon@aporia.com",
4   "project_name": "My Model",
5   "project_slug": "{{
    cookiecutter.project_name.lower().replace(' ', '-')
  }}",
6   "module_name": "{{
    cookiecutter.project_name.lower().replace(' ',
    '_').replace('-', '_') }}",
7   "project_short_description": "My awesome model!",
8   "version": "0.1.0"
9 }
```

These are the variables that the user can easily change when he clones the template.

Next, search for all occurrences of my_model and change them to:

```
{{ cookiecutter.module_name }}
```

This works in both file names and file content.

The user can then clone the model template by running:

```
cookiecutter <your-github-organization>/model-template
```

What's missing from this?

Well... a lot actually!

Here's a partial list:

- HTTPS & Authentication
- Environments (staging, production)
- Common library for preprocessing, postprocessing, etc
- Model input & validation
- Training orchestration
- and probably much more!

Thank you for reading!

If you're interested in this project, make sure to star the [GitHub repo](#) 🙏

Summary

Congrats! You've made it this far which means that you now have the knowledge and tools needed to build a basic ML platform. Together, we built a simple Machine Learning infrastructure, based on open-source tools. We leveraged DVC to perform data versioning and used MLFlow for experiment management and packaging. We built model servers based on FastAPI, and understood how to monitor them for drift using Aporia's ML Observability solution.

The entire build was done on top of AWS, but theoretically, you're not limited to it. You can use Azure, Google Cloud, or any other cloud provider. It's important to note that when building your own ML platform, you should NOT take these tools for granted. You should evaluate alternative tools to find ones that can be customized to your specific use case and models. One way we've tried to make this process a little easier is by creating [MLOps.toys](#) – a curated list of the most useful MLOps tools and projects for training orchestration, experiment tracking, data versioning, model serving, model monitoring, and explainability. You can use MLOps.toys to begin evaluating the right solutions for your ML infrastructure.

While creating your ML infrastructure and deciding on the best tools, it's important to begin prioritizing their contribution to the success of your ML systems. At the end of the day, you're building this ML infrastructure to be able to build, train, and deploy models at scale, while ensuring your organizations' models are performing their best and driving their intended results. A key part of that infrastructure is having a way to monitor and explain machine learning models once they hit production. Just like other tools you need for your ML infrastructure, finding the right solution for ML Observability is essential for your organization.

By implementing a dedicated ML Observability solution with ML Monitoring, Explainability, and Investigation into your ML pipeline, you'll be taking an important step to putting Responsible AI into practice.

If you're ready to explore ML Observability solutions, we welcome you to test out Aporia's Cloud-Native ML Observability using our [free community edition](#), or [request a demo](#) to see Aporia's customizable machine learning monitoring and explainability in action.



CLOUD-NATIVE ML OBSERVABILITY

www.aporia.com