MLflow Setup on Kubernetes with RDS and S3

Author: Smit Kiri

Claps: 71

Date: Dec 12, 2022

Iâ€™m a Machine Learning Engineer at Klaviyo on our Conversations AI team. A few months ago, I found myself struggling to keep track of my experiments. I needed a central place to store experiment details such as training data, hyperparameters, metrics, and models. Other data scientists here were tracking their experiments with google sheets and with multiple Jupyter notebooks. After a point, this gets unmanageable and itâ€™s hard to compare experiments. So I looked for a better solution and, with agreement from colleagues, selected the open source platform MLflow.

We decided to deploy MLflow on our Kubernetes cluster, and although I had previous experience with MLflow, this was my first time working with Kubernetes. It took time to understand Kubernetes and find my way around different AWS components, which is why I wanted to share what I learned in this post.

MLflow Tracking allows data scientists to track model training. It records hyperparameters, metrics and artifacts such as datasets and model files. This is especially useful for maintaining multiple projects, and multiple experiments within a project. It also makes it easy to share results with other team members.

In this guide, Iâ€™ll walk you through Kubernetes basics and help you deploy MLflow Tracking on AWS EKS (Elastic Kubernetes Service).

Kubernetes Basics

Youâ€™ll need to know some basic Kubernetes concepts and terminology for this setup.

Kubernetes Cluster: Weâ€™ll be deploying MLflow to a cluster. A cluster runs multiple containerized applications. Details of the many components in a cluster arenâ€™t important for this guide. Think of this as the top level.

Pod: A pod is the smallest computing unit in Kubernetes. It has a group of one or more containers that share storage and network resources, and a specification for how these containers should be run.

Namespace: A namespace in a Kubernetes cluster provides a way to group resources together. This isnâ€™t required, but is useful if you have multiple teams using the same cluster.

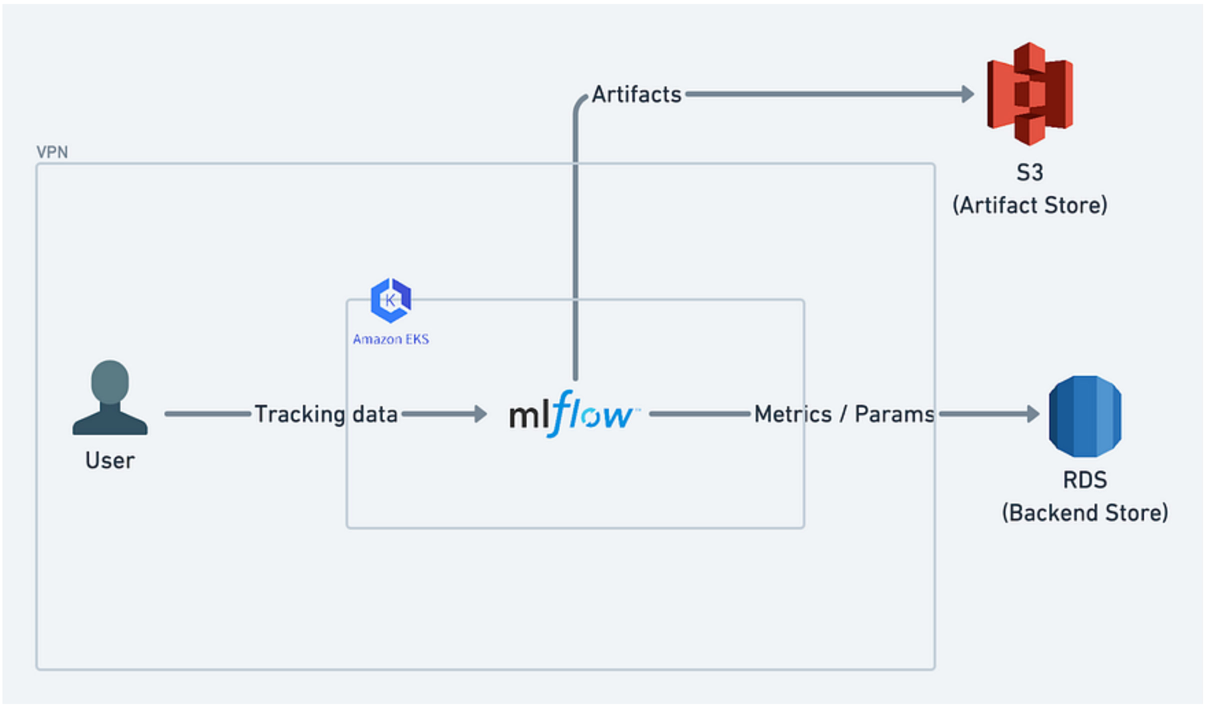
ConfigMap: This is used to store non-confidential data in key-value pairs. Pods can use these as environment variables.

Deployment: A deployment defines how your application is run. This will make more sense below.

Service: While a deployment defines how an application is run, it doesnâ€™t define how the outside world communicates with the application. A service defines that.

MLFlow setup on Kubernetes

Now that we have a basic understanding of Kubernetes, letâ€™s set up MLflow! MLflow has a simple architecture, as shown in the figure below. It has three main components: a backend store that stores experiment metadata (like hyperparameters and metrics), an artifact store that stores all artifacts (like model files), and the MLflow server which has an API and a UI to view and record all this information.



Weâ€™ll be using a MySQL database as the backend store (Amazon RDS) and an S3 bucket as the artifact store.

Setting up the cluster

Before we get started, make sure you have kubectl set up (instructions) and an EKS cluster set up (instructions).

Run the following command to select your cluster:

```
kubectl config use-context <cluster-name>
```

Working with Kubernetes involves working with a lot of YAML files. First create the namespace file that will be used to group all MLfow resources.

mlflow_namespace.yaml:

```
apiVersion: v1
kind: Namespace
metadata:
  name: mlflow
```

Then, to create the namespace in the cluster, run:

```
kubectl create -f mlflow_namespace.yaml
```

Setting up stores and permissions

Create a database in RDS and an S3 bucket to act as the backend and artifact stores respectively. (At Klaviyo we use Terraform and I followed these guides: RDS Setup and S3 setup.)

Make sure that the EKS cluster is able to access your database and S3 bucket. In our case, we did this by creating a role with appropriate policies for EKS, RDS and S3. We have automation to configure this at Klaviyo, but the resulting policy was something like:

Permissions policy:

```
{
    "Statement": [
        {
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:ListBucket",
                "s3:DeleteObject"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:s3:::klaviyo-mlflow"
            ]
        },
        {
            "Action": [
                "rds-db:connect"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:rds-db:<region>:XXXXXXX:dbuser:cluster-XXXX/mlflo
            ]
        }
    ],
    "Version": "2012-10-17"
}
```

Trust policy:

```
{
 "Version": "2012-10-17",
 "Statement": [
     {
         "Sid": "AllowEKSProvider",
         "Effect": "Allow",
         "Principal": {
             "Federated": "arn:aws:iam::xxxx:oidc-provider/oidc.eks.<regio
         },
         "Action": "sts:AssumeRoleWithWebIdentity",
         "Condition": {
             "StringEquals": {
                 "oidc.eks.<region>.amazonaws.com/id/xxxx:sub": "system:se
                 "oidc.eks.<region>.amazonaws.com/id/xxxx:aud": "sts.amazo
             }
         }
     }
 ]
}
```

Link this role to a service account on EKS.

mlflow_service_account.yaml (replace with ARN of your role):

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: mlflow
  namespace: mlflow
  annotations:
    eks.amazonaws.com/role-arn: arn:aws:iam::999999:role/k8sAssumableRole
automountServiceAccountToken: true
```

To create this service account, run:

```
kubectl apply -f mlflow_service_account.yaml
```

(You can also choose to not use RDS, S3 or both, and have all the data live on the EKS cluster.)

Building the Docker Image

Create a docker container that will run the service. Youâ€™ll need the following files:

requirements.txt:

```
mysqlclient
boto3
mlflow
```

entrypoint.sh (adjust region as appropriate):

```
#!/bin/sh
SSL_CA_PATH=/opt/program/us-east-1-bundle.pem
wget -O $SSL_CA_PATH https://truststore.pki.rds.amazonaws.com/us-east-1/us

BACKEND_STORE_URI="mysql://$MLFLOW_DB_USER:$MLFLOW_DB_PASSWORD@$MLFLOW_DB_

mlflow server \
  --host 0.0.0.0 \
  --port "$MLFLOW_PORT" \
  --artifacts-destination "$MLFLOW_ARTIFACT_URI" \
  --backend-store-uri "$BACKEND_STORE_URI" \
  --serve-artifacts \
  --default-artifact-root mlflow-artifacts:/mlruns/
```

Dockerfile:

```
# syntax = docker/dockerfile:1.3
FROM python:3.9

COPY . /opt/program/mlflow
WORKDIR /opt/program/

ENV MLFLOW_PORT 8975

RUN pip install -r requirements.txt

EXPOSE ${MLFLOW_PORT}

RUN ["chmod", "+x", "entrypoint.sh"]
ENTRYPOINT ["entrypoint.sh"]
```

Build the docker image and upload it to ECR (Elastic Container Registry) using the following commands.

First, create a repo in ECR:

```
aws ecr create-repository --repository-name mlflow
```

You should see the ECR URI in the output. (You can also do the above step in the AWS console.)

Then, build and tag the docker image:

```
docker build -t mlflow:latest .
docker tag mlflow:latest <paste-ecr-uri>:latest
```

Push the docker image to ECR:

```
aws ecr get-login-password | docker login --username AWS --password-stdin
docker push <paste-ecr-uri>:latest
```

Deploying to EKS

Create a configmap with our database and S3 bucket information.

mlflow_configmap.yaml:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mlflow
  namespace: mlflow
data:
  MLFLOW_DB_HOST: db.cluster.uri.rds.amazonaws.com
  MLFLOW_DB_USER: mlflow_user
  MLFLOW_DB_PORT: "3306"
  MLFLOW_DB_NAME: mlflow_backend_store
  MLFLOW_ARTIFACT_URI: "s3://mlflow-artifact-store"
```

Add the configmap and the database credentials:

```
kubectl create configmap mlflow mlflow_config.yaml
kubectl create secret generic mlflow-db-credentials --from-literal=db_pass
```

Weâ€™re ready to deploy MLflow. Youâ€™ll need a deploy YAML file.

mlflow_deploy.yaml:

```
apiVersion: apps/v1
kind: Deployment

metadata:
  name: mlflow-tracking-server
  namespace: mlflow
  labels:
    app: mlflow-tracking-server

spec:
  replicas: 1
  selector:
    matchLabels:
      app: mlflow-tracking-server-pods
  template:
    metadata:
      labels:
        app: mlflow-tracking-server-pods
    spec:
      serviceAccount: mlflow
      containers:
        - name: mlflow-tracking-server-pod
          image: <paste-ecr-uri>:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8975
          envFrom:
            - configMapRef:
                name: mlflow
          env:
            - name: MLFLOW_DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mlflow-db-credentials
                  key: db_password
                  optional: false
```

spec.replicas define how many replicated pods should be created.

spec.template defines the pod template â€” the metadata for the pod and what containers should live on the pod.

spec.selector defines how the deployment will select which pods to manage. Here, it will manage the pods that have the label *app: mlflow-tracking-server-pods* which is defined in spec.template.metadata.

We set the containerâ€™s environment by using **envFrom**, which gets the environment variables from the configmap that we defined earlier. Additionally, we set an environment variable containing the database password, which we retrieve from **mlflow-db-credentials** we set above.

Deploy to the cluster:

```
kubectl apply -f mlflow_deploy.yaml
```

Now you should be able to see your pod(s) running on the cluster with this command:

```
kubectl get pods -n mlflow -o wide
```

Creating the EKS Service

We have our pod(s) up and running but there is no way to communicate with them. Weâ€™ll need to define a service which creates load balancer(s) for us to be able to access the MLflow server. We have two use cases here: to access the web UI over the VPN, and to access the API from internal AWS resources like SageMaker notebooks.

EKS supports two different load balancers, an internal and an external load balancer for these two use cases. We can define those in a YAML file.

mlflow_service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: mlflow-tracking-server
  namespace: mlflow
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags:

spec:
  loadBalancerSourceRanges:
    - "0.0.0.0/0"
  type: LoadBalancer
  selector:
    app: mlflow-tracking-server-pods
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8975
      name: http
    - port: 443
      targetPort: 8975
      name: https
---
apiVersion: v1
kind: Service
metadata:
  name: mlflow-tracking-server-internal
  namespace: mlflow
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-internal: "true"

spec:
  type: LoadBalancer
  selector:
    app: mlflow-tracking-server-pods
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8975
      name: http
    - port: 443
      targetPort: 8975
      name: https
```

Here, **loadBalancerSourceRanges** allows us to specify which external IP addresses can access the MLflow server. Above we show **0.0.0.0/0** which will expose it to the wider internet. In our actual configuration, we specify the IPs of our VPN.

As with the deployment YAML file above, **spec.selector** defines how the service chooses the pods to manage.

Create these load balancers:

```
kubectl apply -f mlflow_service.yaml
```

See the external / internal URLs to access MLfLow:

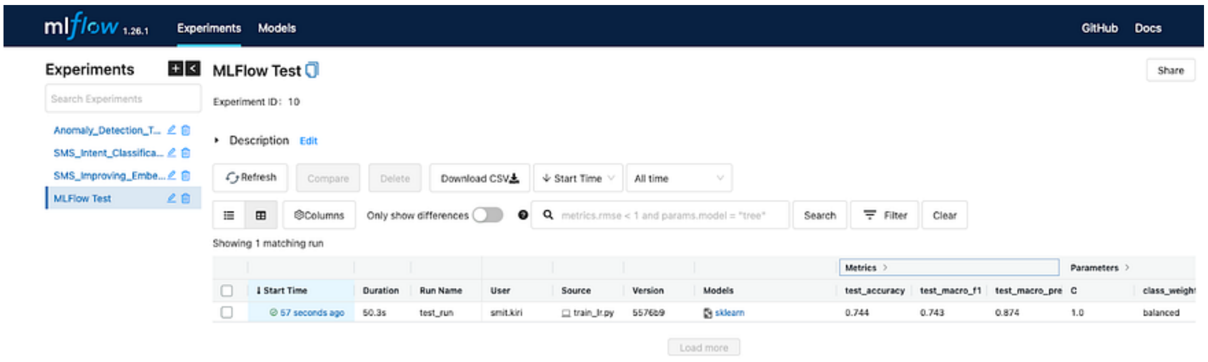
```
kubectl get svc -n mlflow -o wide
```

Trying it out!

Hereâ€™s a simple example. I train a logistic regression model on a SageMaker notebook instance and track the run on MLFlow using the auto-logging feature. I use the internal URL here because my notebook is running inside our AWS account.

```
mlflow.set_tracking_uri(â€œ<paste-load-balancer-url>â€�)
mlflow.set_experiment(â€œMLflow_Testâ€�)
mlflow.sklearn.autolog()

mlflow.start_run()

<my-training-code>

mlflow.log_metrics({"test_accuracy": test_acc, "test_macro_f1": test_macro
mlflow.log_artifacts("/path/to/data/dir/")

mlflow.end_run()
```

And you can access the MLflow UI via the external URL from your web browser. Learn more about tracking your experiments here!



MLFlow Test

Experiment ID: 10

Description Edit

Showing 1 matching run

							Metrics			Parameters	
Start Time	Duration	Run Name	User	Source	Version	Models	test_accuracy	test_macro_f1	test_macro_pre	C	class_weight
57 seconds ago	50.3s	test_run	smit.kiri	train_lr.py	5576b9	sklearn	0.744	0.743	0.874	1.0	balanced