MLOps Scaling Machine Learning Lifecycle in an Industrial Setting

Yizhen Zhao, Adam S. Z. Belloum, Gonçalo Maia da Costa, Zhiming Zhao

Abstract—Machine learning has evolved from an area of academic research to a real-world applied field. This change comes with challenges, gaps and differences exist between common practices in academic environments and the ones in production environments. Following continuous integration, development and delivery practices in software engineering, similar trends have happened in machine learning (ML) systems, called MLOps. In this paper we propose a framework that helps to streamline and introduce best practices that facilitate the ML lifecycle in an industrial setting. This framework can be used as a template that can be customized to implement various machine learning experiments. The proposed framework is modular and can be recomposed to be adapted to various use cases (e.g. data versioning, remote training on Cloud). The framework inherits practices from DevOps and introduces other practices that are unique to the machine learning system (e.g.data versioning). Our MLOps practices automate the entire machine learning lifecycle, bridge the gap between development and operation.

Keywords—Cloud computing, continuous development, data versioning, DevOps, industrial setting, MLOps, machine learning.

I. Introduction

ARTIFICIAL Intelligence (AI) and Machine Learning (ML) are projected to become the mainstream technologies in the coming years. Machine Learning offers powerful toolkit for solving complex real-world problems, either in the academic research work or in industrial innovation.

A. Background

Machine learning in academic research and in the real-world machine learning system, may sometimes mean different things. According to Sculley et al. [1] ”machine learning code is only a small fraction in the real-world machine learning system”. The required surrounding infrastructure is complex and usually, it takes longer to deploy ML in production[1], compared to developing ML models. MLOps, or DevOps for machine learning, is becoming a necessary skill set for enterprises to leverage the benefits of ML in the real world. It is a practice for better collaboration and communication between the data scientists and data engineers to improve the automation of the entire machine learning lifecycle and deploy it in the production environment.

This paper takes a machine learning project at Dashmote [2] as the example use case and develops a prototype that applies MLOps practices to the machine learning lifecycle. It helps us to tackle the problems and challenges when applying ML in an industrial environment and improves the ML lifecycle management process, which includes model and data versioning, keeping track of experiment results so that the machine learning project has reproducibility and traceability, model monitoring and model deployment in production.

The rest of this paper is organized as follows. In Section I, we present the background information about machine learning in production and outline the generalized problem statement. In Section II, we will describe the state-of-art relevant to this paper. Then, our ML lifecycle process and MLOps architecture design are introduced in Section III, followed by the details of each part within this prototype. In Section IV, the discussion and future work are presented. The summary of this paper is in Section V.

B. Problem Statement

In [3] we have conducted a literature review on the usage of machine learning systems in production. This review identifies the challenges and difficulties in building and maintaining ML systems in a production environment, and points out how MLOps can help to solve some of these challenges. In this section, we highlight some of the problems and challenges facing the introduction of the machine learning approach as they are relevant to the work presented in this paper.

Machine learning in production is not only about implementing ML models but also building the required infrastructure [4]-[6]. As machine learning development is an iterative process, data scientists often need to do multiple experiments to optimize a metric (e.g. prediction accuracy). This might lead to hundreds of different versions of ML algorithm code, data, hyper-parameters or experiment results. Managing all the versioning, results turns out to be a big problem. In addition, the ML lifecycle is composed of many steps. It is difficult for ML development teams to manage the entire ML pipeline and deliver good quality products in a short time without automation tools and centralized repositories where they keep the ML artefacts, the experiment results, etc. ML systems in the production environments face the same challenges as traditional software services, and some are ML-specific challenges. DevOps introduces a set of practices in engineering that focuses on techniques and tools

Yizhen Zhao and Zhiming Zhao are with University of Amsterdam, Science Park 904, 10988 XH Amsterdam, Netherlands (e-mail: yizhenzhao@hotmail.com, z.zhao@uva.nl).

Adam S.Z. Belloum is with the Netherlands eScience Center, Science Park 402 (Matrix III) 1098 XH Amsterdam, The Netherlands (e-mail: A.S.Z.Belloum@uva.nl).

Gonçalo Maia da Costa is with Dashmote B.V, Rokin 86, 1012 KX Amsterdam (e-mail: goncalo.costa@dashmote.com).

[1]By production we mean an environment where code/apps/services are deployed and available for users.

to maintain and support existing production systems. While there is no agreed best practice to handle the whole machine learning lifecycle in production. The above problems are also encountered in Dashmote use case, when we are trying to run ML in the industrial setting.

II. Related Work

In this section, we briefly discuss some relevant research work which enables us to clearly highlight the added value of the proposed framework. Based on the research work, we identified some tools or services that can be used in this use case and practices to build our own MLOps framework, which will be introduced in Section III. Here we classify the selected literature based on the topics below.

Data Version Control: As mentioned in Section I-B, iterative experiments result in different versions of data data, code, parameters, etc. *(1)* Inspired by Git [7], Anant introduced two tightly-integrated systems in their paper [8], an open source tool [9]. One is the dataset version control system (DSVC) that enables data scientists to capture their modifications, identify different versions and share datasets. Another one is DATAHUB, a platform built on top of DSVC, which allows data scientists to perform data analysis, data cleaning, and visualization. DSVC is similar to git but supports richer query languages and has more features. For instance, SQL-based querying and analyzing for specific versions of dataset. There are some other competitive data versioning tools [10]: *(2)* DVC, data version control, will be introduced and used in this use case (Section III-D). *(3)* Pachyderm [11], a data science and processing platform with built-in data versioning and lineage. It deals with plain text, binary files and large datasets. A centralized repository exists and your data is continuously updated in the master branch of the repository. You can work with a specific data commit in a separate branch. *(4)* AWS Sagemaker Ground Truth [12], a data labeling service that provides accurate training dataset for machine learning. It provides a custom or built-in data labeling workflow that allows you to create your own labeling job and then the user can use the labeling interface to label the dataset.

Model Deployment: ML models can only deliver added value to an organization when they are available to users or other systems and deployed in production. There are different ways of deploying models. *(1)* AWS Sagemaker [13] is a fully managed machine learning service and a reliable way to deploy the ML models into production quickly. Using AWS Sagemaker for model deployment is also our choice in this use case, described in Section III-E. *(2)* Azure [14] provides a full MLOps cycle for machine learning projects. It provides its own way of deploying ML models [15]. The workflow is similar no matter where you deploy your models. First is to register the model that you want to deploy. Secondly, the code, an entry script, which will be used in the web service, for performing the predicting on input data. Finally is to define an inference configuration which describes the Docker container and all the files within your project source directory to use when deploying the web service. *(3)* MLflow [16], an open source for ML lifecycle management, also supports model deployment. MLflow is also introduced in this project, but our main focus is on MLflow tracking (Section III-G). Some features of MLflow, such as MLflow Model Registry, Python APIs (e.g.`mlflow.sagemaker`) can be used together to deploy the model to custom serving tools. Model register and version control can be done with MLflow Model Registry. `mlflow.sagemaker` [17] works similarly as our Sagemaker batch transform [18] (in Section III-E).

MLOps Framework: *(1)* Emmanuel [19] introduced an edge MLOps framework for edge Artificial Intelligence Internet of Things (AIoT), which is a system that facilitates edge computing[2] for AIoT applications. This MLOps framework enables continuous delivery, development and monitoring of ML models at the edge for AIoT applications. Azure machine learning [20] is used for managing ML lifecycle, starting from continuous fetching data from edge devices into Cloud storage. Then data versioning, model training, evaluation. Finally packaging and registering the model and waiting for deploying to edge devices (i.e. in production). Azure DevOps [21] is used to maintain and version control the ML algorithm code used for building ML models, and then build and release ML artefacts, models to edge devices and perform needed jobs. *(2)* A case study is introduced in this paper [22], where the authors present a prototype of MLOps pipeline in KubeFlow Cloud native environment. The workflow triggers container level Cloud-native architecture based on the repository. Data preparation is the first and most time-consuming step, it includes preprocessing, data digestion, etc. Then the model building happens in dedicated PODs[3]. After evaluation and model selection, the final model will be deployed to the REST[4] endpoint. The acceptance of MLOps allows rapid research loops, therefore, the pipeline and model can be arranged in production efficiently.

III. Project Implementation

We first analyze Dashmote current ML development process and describe the MLOps architecture which helps us automating the model development and deployment. The resulting ML development process is a template workflow that can be customized and used for different use cases. In the rest of the section, we describe the implementation and results. In each section, we compare ML development process before and after introducing the proposed MLOps workflow. More details of the MLOps Workflow can be found in original paper [23].

A. Requirement Analysis

Fig. 1 presents an overview of the ML development lifecycle in Dashmote use case. The ML use case in this paper is called flag-combo. It is a classification task, which tells whether a meal is a combo meal[5] or not.

[2]Edge computing is the process of performing computing tasks physically close to devices, rather than in Cloud.

[3]The most basic deployable objects in Kubernetes. A single instance of a running process in your cluster.

[4]REpresentational State Transfer, is an architectural style for providing standards between computer systems on the web.

[5]A combination meal, often referred as a combo-meal.

The emphasis on the current use case is on the last four steps, model development and deployment. The prototype of solving the encountered challenges within this ML lifecycle and implementation details are explained in following subsections. The first step is to *Understand the business requirement*, what are the behaviours we expected from this model and what features are needed. The most time-consuming parts are the *Data Preparation* and *Data Labeling*. Meal data is collected from various online food delivery platforms (e.g. Ubereats [26]). Data needs manual labeling first based on the definition of combo meal we defined. In *Feature Engineering*, *Model Training* and *Model Evaluation*, data scientists start working on model building, training and evaluation. During these three steps, different versions of data and ML artefacts (e.g. ML models) are generated by this iterative process. As mentioned in Section I-B, the lack of data, ML artefacts version control and standardized development process, no automated development flow are the problems we encountered. However, these problems have been taken into consideration in this use case, in the following subsections. In this paper, we will not explain the implementation details of this classification model as it is not the main focus. *Model Deployment*, data engineers are also responsible for integrating the ML systems into current industrial settings. Building an architecture that manages the whole ML lifecycle with the goal of automating the whole process. No standardized model deployment process is also problematic in Dashmote, within this use case, we also proposed a standardized way of deploying ML models into production in the following subsections.

B. MLOps Architecture Design

The overall design of our MLOps architecture is shown in Fig. 2. It also describes how we handle the model development and deployment process in our industrial settings. Before introducing this MLOps framework, there was no mechanism to manage the whole ML lifecycle. No dataset and ML models version control. No standard way of model release and deploy trained models into production. After this MLOps framework is built, with the help of Git [7], DVC [27] and MLflow [16], the dataset and all ML artefacts (e.g.model) are carefully tracked and version controlled. The ML experiment process is standardized and results are kept in a centralized place. Two cloud providers are introduced to scale the model training process and apply prediction in the production environment efficiently. No manual handover is required for model deployment as this MLOps framework bridges the gap between development and operation.

This architecture is composed of two main parts: ML model development (*Step 1, 2, 3, 4*) and model deployment (*Step 5, 6*). Model development starts on the left, from the git repository, *dash-ml-flag-combo*, which contains all necessary ML algorithm code for developing the model. Within this git repository, DVC is used to keep track of the data we used in ML project (*Step 1*). Data version control with DVC is located in Section III-D. After that, building ML models, generating features, and tuning parameters (*Step 2*). We provide the options to execute ML jobs (e.g.model training) in local environment or leverage Cloud resources (i.e.Azure Machine Learning), in case the ML job needs extra compute power. Then, the experiment results (e.g.metrics) and other ML artefacts (e.g.models) are tracked by MLflow tracking, a centralized place where all the experiment results are kept (*Step 3*). MLflow tracking is located in Section III-G. After finishing the iterative experiment process, we use DVC to version control the final models and any other ML artefacts. Finally, using Git to release a new version for that model in that specific state (*Step 4*). All the dataset, models and ML artefacts are stored on Cloud, as long as they have been tracked by DVC and/or MLflow tracking. During the development, Jenkins [28] is used for code unit testing, automatic building and continuous deployment.

After obtaining the trained model and artefacts that will be used in production, the deployment process happens. *dash-docker-flag-combo*, another git repository where we prepare the docker image with prediction logic and inference code that will be used in AWS Sagemaker to start the batch transform job and apply the prediction on unseen data and finally store the transformed data into AWS S3. The model and other artefacts (if any) used in production are loaded during runtime via DVC API because of *Step 4*. Model deployment is located in Section III-E. Since models and other artefacts are tracked by DVC using DVC file [29], and DVC files are version controlled by Git, therefore, DVC API allows you to access the models that under DVC and Git control in repository *dash-ml-flag-combo*. Airflow [30] is used here to help us automatically trigger the model deployment pipeline.

C. Refine ML Development Process

In this section, we describe the details of our ML development workflow. The Git repository structure and details can be found in the original paper [23], with brief introduction of essential components. Here we visualize the relationship of each component into the following workflow, Fig. 5. We use python scripts here to represent each step. The python script can contain a python class, or a set of python functions. Within our current strategy, different versions of datasets and models are safely tracked by DVC. All the experiment runs/results are collected in a centralized place, provided by MLflow. They are easy to manage and trace. ML development can be done in different environments, either in the local environment or on Cloud. Git is responsible for code version control and code, model release. These three tools work together to guarantee the traceability and reproducibility of ML projects. Jenkins helps in continuous integration and continuous development, it runs autonomous tasks we defined, such as code unit testing, data testing, docker image build, etc. The end-to-end developing workflow is described in the following steps:

- If you are working with a new Git repository without any DVC setup, the first step is to install DVC and initialize it in your Git repository. Otherwise, you can start following the Gitflow [31] to create your own feature branch[6].

[6]A branch where you used for developing new features

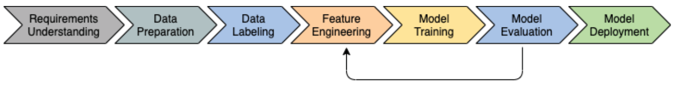


Fig. 1 Proposed ML Lifecycle

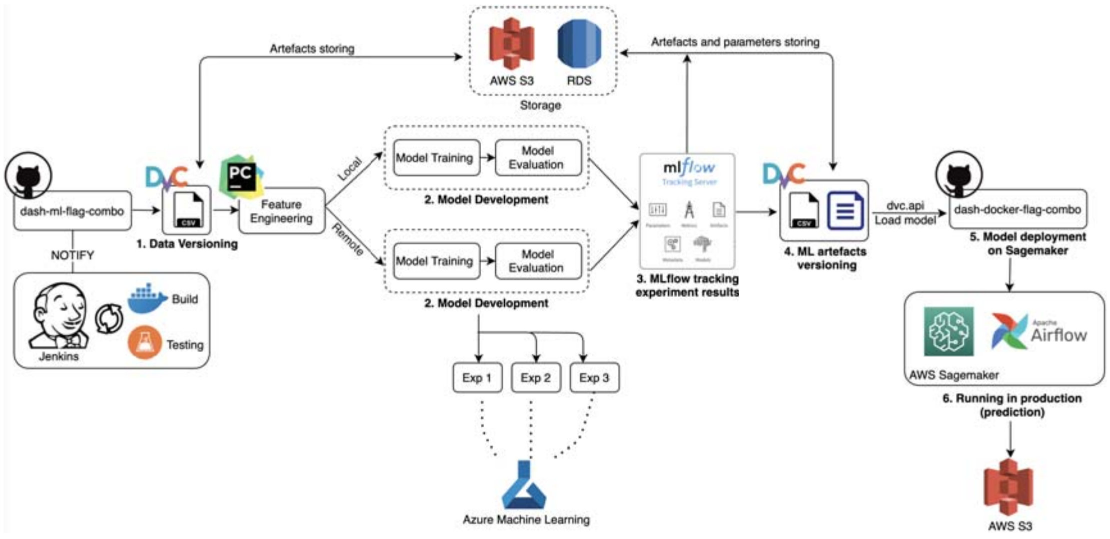


Fig. 2 MLOps Architecture

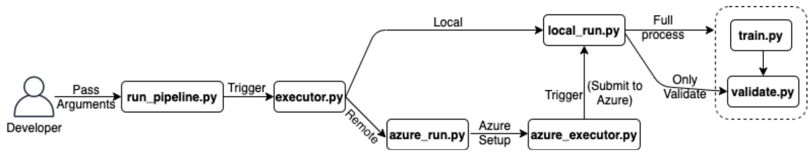


Fig. 3 ML development internal workflow

Use `dvc pull` to synchronize the datasets and other artefacts, download them from remote storage.

- Use `dvc add` to add dataset into DVC control if you are adding new datasets or updating datasets. For example, `dvc add assets/data/NL/NL_training.csv` will generate the corresponding DVC file for versioning the NL training data.
- Create country-based[7] DVC pipelines that run each stage of the ML project. We create two base stages: full process stage (i.e. train and validate) and only validation stage because there might be a case that we only want to re-validate the model (e.g. a new validation set is added). This country-based DVC pipelines provide a clear classification between different sub-projects. The example code of creating a DVC pipeline in this case is presented in Appendix A-A, the generated DVC files (i.e. *dvc.yaml*, *dvc.lock*) are presented in Appendix A-B.
- Triggering the ML pipeline process by either executing Python script or using DVC command `dvc repro`. The process starts by the user, passing necessary arguments to `run_pipeline.py`. Then it triggers the `executor.py` for executing ML pipeline process either in local environment or in Cloud environment based on the arguments user passed in. The above DVC-related information can be found in Section III-D.
  – Local environment: The local pipeline (i.e.`local_run.py`) is run. Then it triggers either a full process or only validate the process.
  – Cloud environment: The azure pipeline (i.e.`azure_run.py`) is run. It creates all necessary configuration for executing ML jobs on Azure, then the `azure_executor.py` will kick off the local pipeline with this configuration on Azure ML. It is always the local pipeline (i.e.`local_run.py`) that is run, since both environments are using the same model building python scripts, Azure ML just requires different settings. Configure ML jobs to Azure ML is described in Section III-F.
- During each experiment run, MLflow tracking (Section III-G) logs all the necessary results, such as metrics, parameters used, even models.
- After finishing model building and receiving satisfying results, we use DVC command `dvc add` to version control the final version of ML artefacts (e.g. models), as well as the data we used that the model has been trained on.
- Before letting Git to version control everything, using `dvc commit`, `dvc push` to commit DVC pipelines, DVC files to DVC and push dataset, artefacts to remote storage. Then let the Git version control all the changes

[7]In our use case, we have datasets for different countries.

and results.

- At this point, a pull request[8] can be opened for team members to review. Once the *feature* branch[9] is merged and a new release created, a ML development process has been completed.

D. Data Versioning with DVC

As mentioned in Section I-B, data versioning is a big challenge in a machine learning project. Arising questions are as follows: How to manage the different versions? How to guarantee the traceability and reproducibility of ML projects?

DVC [27] is a tool that makes ML models shareable and reproducible. We choose DVC because firstly it works similarly as Git, providing easy-to-use commands and can handle large files or models. Secondly, it can easily integrate with Git-based projects, work together with Git to help us version control the dataset and the code. Last but not the least, it is an open source tool. Before there was no mechanism to version control the dataset that has been used in ML projects. Dataset we used for training some legacy models no longer exists, which makes us lose the reproducibility of certain models. Now with this data versioning mechanism, the dataset has been taken care of by DVC and kept on remote storage.

We summarize the core idea of using DVC with data versioning in this section. It builds the foundation for this use case as it solves one of the biggest challenges, data versioning in machine learning projects. The implementation details can be found in the report, online blog and original paper [23]-[25]. In this paper, we extend the work to integrate the data versioning and workflow mechanism we implemented before with other services.

In order to use DVC and DVC features properly within ML projects, the first step is to install and initialize it. Then assign a remote storage (e.g. AWS S3) to DVC, where the datasets will be stored instead of keeping them in the Git repository. DVC uses a so-called ***.dvc** file [29], which contains a unique md5 hash that uniquely identifies your data files, to help you version control the data files. Then Git is responsible for version control of the code and that DVC file. DVC has several features:

- Versioning data: It provides a simple command, `dvc add`, to add data files into DVC control and generate the ***.dvc** file.
- Remote storage: DVC supports several remote storage (e.g. AWS S3). Data files can be stored on remote storage if a remote storage is assigned.
- Retrieve data files: Having a DVC-controlled data file stored remotely on the Cloud, it can be downloaded to a local project when needed.
- Building ML pipelines: DVC also supports building DVC-based ML pipelines which allows you better organize projects and reproduce the workflow and results later. It uses *stage* to represent each single data process. There are two files: *dvc.yaml* file [32], in which one or several stages are presented (e.g.training stage). Inside each stage, it specifies its dependencies (e.g. input data file), outputs (e.g. expected output model file) and commands that are used to run the script. Once the stages are presented in the dvc.yaml file, it can easily be reproduced by simple DVC command. And *dvc.lock* [33] file, it helps to record the state of the ML pipeline(s) and track its outputs by using md5 hash. Creating a DVC-based ML pipeline is mentioned in Section III-C and code examples are listed in Appendix A-A and A-B.

[8]An notification to team members that a developer completed a feature and request team members to review the work.

[9]A branch where you used for developing new features.

E. Model Deployment with AWS Sagemaker Batch Transform

ML model starts providing value to the enterprise when it has been deployed into the industrial production environment. There are lots of tools or services that are aiming to deploy ML models into production in an efficient and scalable way. Amazon Sagemaker [13] is a fully managed service that supports ML model building, training and deploying. One of the features, batch transform [34], is a high-performance and high-throughput method for transforming data and generating inferences. It is ideal for dealing with large datasets.

The implementation details can be found in the same report [24] and the online blog [35]. We leverage AWS Sagemaker batch transform to deploy trained models into production. The idea is using a simple API to run prediction on the large or small batch dataset. Sagemaker provides a set of parameters that allows you to customize your prediction function. For instance, by customizing the payload size of your batch transform job, it will load as much records as possible within that payload size in the dataset and perform prediction on that mini-batch. Within our testing, we utilize an instance with 8 CPU, 32 GiB memory to perform prediction on a 1.8G JSON file. It only takes 18 minutes to finish the prediction and this type of instance costs only $0.461 per hour [36]. By standardizing the model deployment workflow, we have a standard and consistent way of deploying model into production environment. With the help of DVC and Airflow, model deployment can be executed automatically and without any handover process.

F. Productize ML Experiment Process Remotely on Cloud

It is a common thing that ML development first starts in a local environment, then scales out to Cloud environment. One of the advantages of Cloud-based services is that it gives developers access to high-performance infrastructure that they can pay for what they use. Therefore, we introduce the option to utilize Cloud resources, Azure Machine Learning [20], to help us scale out our ML development process.

Azure ML allows you to deploy custom ML jobs to a Cloud-based environment. There is a set of configurations that needs to follow. These six steps work together to provide an environment that ML jobs (e.g.model training) can run on with customized requirements. The main function of each step is described below and the corresponding Python code example for creating each step can be found in the original paper Appendix B.1 [23].

- Azure ML Workspace: A logic container that manages all ML assets, such as compute instances, data storage, pipelines, models, etc. It is the place where your ML jobs happen and the foundation of running ML jobs.
- Compute target: An environment where you can train the ML models on. It provides a variety of resources and developers can choose based on their needs. It is to pay for what you use.
- Experiment: Each ML run or ML job is called an experiment on Azure ML. All the information related to that run will be logged into one experiment, identified by an unique run ID. By having an experiment name for your ML projects, the ML runs can be grouped together by the experiment name.
- Environment: An encapsulation of the environment that includes all the necessary Python libraries and packages that will be used in ML jobs. Azure ML provides curated environments but also you can customize it by using your own docker image as the environment.
- Configure script run: We already have above settings, then is to connect them all together and tell Azure ML what is the configuration, where is the project folder that includes all necessary files and scripts and where are the Python scripts that execute the ML job (e.g model training).
- Submit the Experiment: With all the steps above finished, the final step is to actually submit the ML job to Azure and do the ML job.

G. Productize MLflow Tracking Server

MLflow is an open source for managing machine learning lifecycle. It allows tracking ML experiments, guaranteeing the reproducibility of ML projects. Also including model deployment and model registry. In this use case, we only use **MLflow Tracking**. The interface of MLflow Tracking and Model Registry is displayed in Appendix B. One of the biggest challenges for us is lacking a centralized place and a strategy to manage all the experiment results. Without MLflow Tracking, our ML projects may require manual logging or use Git to version control experiment results, which is not very user friendly and error-prone. The way MLflow tracking works is by recording MLflow runs or experiments into either local files, a SQLAlchemy[10] compatible database or a remote tracking server [37]. There are two components used for storage: **backend store**: for storing MLflow entities (experiment runs, parameters, etc) and **artifact store**: for artifacts (models, files, etc). We choose the scenario where the tracking server, backend and artifact store are hosted remotely (i.e. MLflow with remote tracking server). The advantages are that team members can access this tracking server if they have permission and the experiment runs on existing MLflow Tracking are shareable.

In this section, we will briefly introduce the Cloud setup for our MLflow tracking server. We start implementing it locally in docker container and then migrate to Cloud. The local implementation details can be found in the original paper [23], under the same section. The Cloud infrastructure of our MLflow tracking server is shown in Fig. 4. The main technologies and services are:

- AWS Route 53: It provides the custom domain name and DNS settings for our MLflow tracking server. It is the entry point for users to access the MLflow tracking server.
- Application Load Balancer: It helps to automatically distribute the incoming traffic across multiple targets, and coordinates the traffic on the road. Then the load balancer directs the traffic to the AWS EC2 target group [38] we specified. After that the target group routes requests to the registered target, which is our MLflow server that runs on AWS ECS. During the process, EC2 security group is used and acts as a virtual firewall for our application. Only authorized IPs can access our MLflow server.
- MLflow tracking server set up: It is similar to our local setup. We still use AWS S3 as the artifact store as it is our main storage service, but we change the database-based backend store from local docker to Cloud, using AWS RDS to provide the relational database service in the cloud. Then changing the backend setup, the database credentials, to the one we created on AWS RDS. This whole setup is packaged into a docker image and pushed to AWS ECR, run on AWS ECS.
- AWS CloudWatch: Collecting logs, metrics and events from MLflow tracking server.

H. Integrate MLflow Tracking and DVC with ML Experiments

DVC helps us to version control the dataset and ML artefacts, MLflow tracking helps to keep all the ML experiment results. To some extent, they can both version control the ML entities(i.e. metrics) and models. The question is how to define the proper roles for them and the corresponding strategies.

In our case, we decided to use DVC as the main version control tool for ML entities and artefacts. We follow the Git release strategy for DVC. MLflow tracking is mainly for tracking and managing each ML experiment. They work together to guarantee the traceability of our ML projects. The strategy we used for ML development workflow regarding this scenario is described in Section III-C.

If the ML experiment runs in a local environment, we can directly use DVC pipelines (in Section III-D) to version control the outputs (e.g.models) by adding them into DVC pipeline *outs*, then they will be tracked by DVC automatically. Finally using Git to version control relevant DVC files. If ML experiment happens remotely on Azure ML, those ML artefacts are exported to Azure ML workspace after the ML algorithms run, in a special folder *./outputs*. We need to download them from Azure ML workspace to local first, and then add them into DVC pipeline *outs*. The Azure ML configuration and download code examples can be found in the original paper, in Appendix B.1 [23].

IV. Discussion and Future Work

The ML lifecycle and MLOps architecture we proposed combined the research work we found in academic areas,

[10]A library that facilitates the communication between Python programs and database.

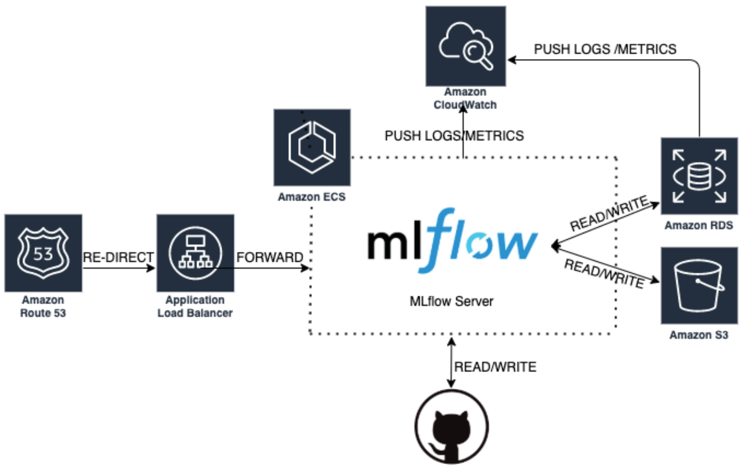


Fig. 4 MLflow tracking server infrastructure

technical tutorials and personal experience. Due to the time limit, some future work can be done to enhance the whole infrastructure. We use several tools and technologies within this MLOps architecture. The decision can be made based on experience and requirements or research work. The discussion and future work are presented in this section.

A. Discussion

We used three different tools or technologies within our MLOps architecture. DVC (Section III-D) is used for versioning control of the datasets. Within our proposed workflow, it can handle the data versioning lifecycle for ML projects. No matter what tools are used, it is important to come up with a workflow that handles your data versioning lifecycle.

For model deployment, we stick with AWS, our main Cloud provider, Sagemaker batch transform (Section III-E). Other approaches like MLflow can be a good candidate, since it supports the whole ML lifecycle management and provides an API to deploy the registered model into downstream services (e.g.Sagemaker). Azure, as another Cloud provider, provides all necessary services to manage the ML lifecycle. If Azure is your main Cloud provider, all the ML-related services can be provided by Azure. We utilize the Cloud computing (i.e. Azure ML) to do the remote ML training. This is the team decision because of the budget but AWS offers a similar option (i.e.AWS Sagemaker), it also supports model training [39]. By providing the docker image, which contains the ML algorithm code for training the model, to Sagemaker training job, and choose the compute resources, Sagemaker can train the model and export the output files to AWS S3. Another option is to run the model training on AWS ECS. A docker image can be created locally with the ML algorithms to train the model. Then push the image to AWS ECR, and create a task definition on AWS ECS. Task definition defines which Docker image to use, how much CPU and memory to use, and where to launch the task (e.g. AWS EC2). ECS makes it easy to deploy and scale Docker containers running applications, services, etc.

B. Future Work

We use a so-called machine learning operations maturity model [40] to evaluate our own MLOps framework. Based on the highlights of different levels in this maturity model and our own circumstances, we almost fulfilled the requirements of level 4. But few parts are missing and can be improved in this use case.

Our remote model training on Azure ML (Section III-F) is triggered manually by a local Python script (i.e.`azurre_executor.py`), which means developers have to wait until the model training is done. This is not ideal for the scenario where the training might take hours to finish. This manual triggering can be automated by adding this action into Jenkins [28] pipeline, our test automation tool used for continuous integration (CI) in software systems. For instance, when the ML job is ready to be deployed into Azure ML, Jenkins can trigger the pipeline and the whole project is packed together and submitted to Jenkins. Then the ML job is executed on Azure ML. During the experimentation, the same MLOps workflow happens. After that, developers can decide whether the results are satisfying enough or not for further decisions.

The CI/CD[11] pipeline, which forces automation in building, testing, training, retraining and even deploying, is missing in this use case. Based on the research work in Section II and our own MLOps framework, it can be designed as follows. A data validation test should be introduced before CI/CD, meaning the data quality should be evaluated first. Then follow the Git workflow, doing experimentation on git *feature* branch[12]. Once a git commit is made, the code unit tests are executed, to make sure the basic function of ML algorithms is working as expected. Once the *feature* branch is merged into the *develop* branch, the CI pipeline triggers, which runs the same code unit tests and remote model training on Azure ML. After that, the CD pipeline runs the model validation on Azure ML as well. Once a milestone is created, a *release* branch is published

[11]Continuous integration, continuous development or continuous deployment.
[12]A branch where you used for developing new features.

which means a new version of the model is released. Then the same CI/CD pipeline is triggered and now the model is ready to be used in production.

V. Conclusion

In this paper, we present the design of our MLOps framework and associated infrastructures that improve the ML lifecycle management process. In the model development process, we utilize DVC for dataset and ML artefacts (e.g. models) version control, Git for code version control and model releases. During the iterative ML development process, a separate server, MLflow Tracking, is used for managing and keeping all ML experiment results (e.g. metrics). They work together to guarantee the reproducibility and traceability of ML projects. Then the flexible options to execute ML experiments either in local environment or using Cloud resources (i.e. Azure ML). In model deployment process, we use AWS Sagemaker Batch Transform to scale the model deployment process in production, with its own batch strategy, customized settings, model is loaded during runtime via DVC API and then apply prediction logic on batch dataset, which is efficient and also affordable. Since we have datasets and ML artefacts (e.g. model) under DVC control in model development process, they can be loaded into production by using DVC API, therefore, no manual handover is needed.

The purpose of this paper is to provide a template framework and workflow strategy for people who are interested in MLOps and trying to build their own MLOps framework. Within this MLOps architecture, a standard ML lifecycle workflow has been established. Data, ML artefacts lifecycle, and the ML experiment details have been carefully tracked and saved. It streamlines and automates the ML lifecycle, reduces the delivery time and labour work, and makes the ML development process more reliable, traceable and scalable.

VI. Acknowledgement

This research has been performed as part of the Enabling Personalized Intervention (EPI) project. The EPI project is funded by the Dutch Science Foundation in the Commit2Data program, grant number 628.011.028.

Appendix A
DVC Code Examples

A. DVC Build Pipeline Example

The code examples are for Section III-C. The full process (i.e. train and validate) stage in the DVC pipeline can be created by following code. It specifies the dependencies (`-d`), outputs (`-o`), metrics (`-M`) and plots (`-plots-no-cache`). Run the command from the directory dvc_pipeline/NL:

```
dvc run train_validate_local \
-d ../../src/dash_ml_flag_combo/scripts/ \
run_pipeline.py \
-d ../../assets/data/NL/NL_training.csv \
-d ../../assets/data/NL/NL_validation.csv \
-o ../../assets/model/NL/NL_flag_combo.pkl \
-M ../../assets/metrics/NL/NL_scoring.json \
-plots-no-cache ../../assets/metrics/NL/ \
NL_confusion_matrix.png \
python ../../src/dash_ml_flag_combo/scripts/ \
run_pipeline.py ${params.country} \
${mode.train_validate} ${pipeline.local}
```

B. DVC Pipeline File Examples

The generated DVC pipeline related files, created by the code above in Section A-A, are presented here, along with the *params.yaml* file.

dvc.yaml file:

```
stages:
  train_validate_local:
    cmd: python ../../src/dash_ml_flag_combo/ \
    scripts/run_pipeline.py ${params.country} \
    ${mode.train_validate} ${pipeline.local}
    deps:
    - ../../assets/data/NL/NL_training.csv
    - ../../assets/data/NL/NL_validation.csv
    - ../../src/dash_ml_flag_combo/scripts/ \
    run_pipeline.py
    outs:
    - ../../assets/model/NL/NL_flag_combo.pkl
    metrics:
      - ../../assets/metrics/NL/ \
      NL_scoring.json:
          cache: false
    plots:
      - ../../assets/metrics/NL/ \
      NL_confusion_matrix.png:
          cache: false
```

dvc.lock file:

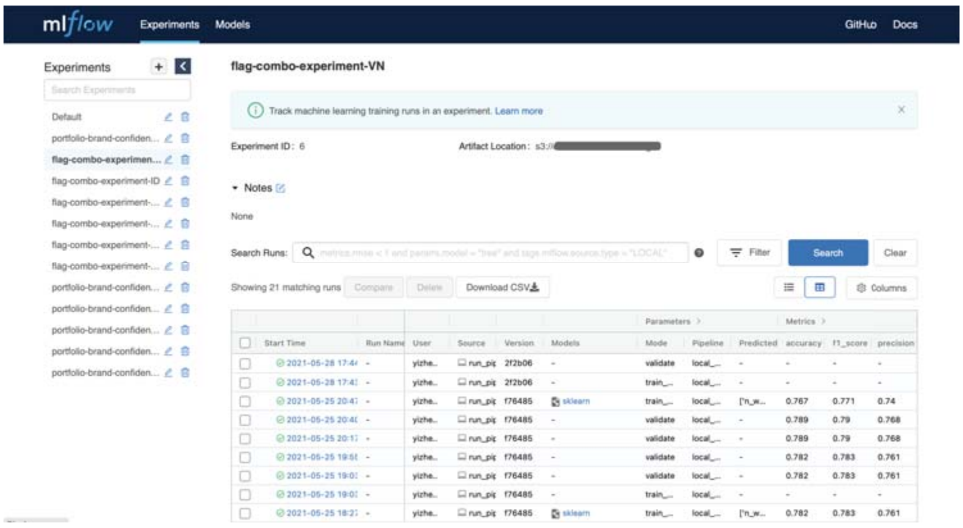
```
schema: '2.0'
stages:
  train_validate_local:
    cmd: python ../../src/ \
    dash_ml_flag_combo/scripts/ \
    run_pipeline.py NL train_validate \
    local_run
    deps:
    - path: ../../assets/data/NL/ \
    NL_training.csv
      md5: 60b27ce834dad76825fn96b26edacc9b
      size: 68595
    - path: ../../assets/data/NL/ \
    NL_validation.csv
      md5: 0c80660e4eaaea3b7337y13de2097506
      size: 35666
    - path: ../../src/dash_ml_flag_combo/ \
    scripts/run_pipeline.py
      md5: ef1658ob652138pd0125d14983de6809
      size: 602
    outs:
    - path: ../../assets/metrics/NL/ \
    NL_confusion_matrix.png
      md5: d60480r7a5c2f9033k534660b5d631eb
      size: 14634
    - path: ../../assets/metrics/NL/ \
    NL_scoring.json
      md5: 5d1ad161l06df2ecdd778d8fab79f760
      size: 128
    - path: ../../assets/model/NL/ \
    NL_flag_combo.pkl
      md5: 48b8516yh274927617a985v8540915ea
      size: 693806
```

params.yaml file:

```
params:
  country: "NL"
mode:
  train_validate: "train_validate"
pipeline:
  local: "local_run"
  remote: "azure_run"
```

Appendix B
MLflow Tracking Server Interface

The details of MLflow tracking interface are shown in Figs. 5-9.


mlflow
Experiments
Models
GitHub
Docs
Experiments
Search Experiments
Default
portfolio-brand-confiden...
flag-combo-experimen...
flag-combo-experiment-ID
flag-combo-experiment-...
flag-combo-experiment-...
flag-combo-experiment-...
flag-combo-experiment-...
portfolio-brand-confiden...
portfolio-brand-confiden...
portfolio-brand-confiden...
portfolio-brand-confiden...
portfolio-brand-confiden...

flag-combo-experiment-VN

Track machine learning training runs in an experiment. Learn more

Experiment ID: 6   Artifact Location: s3://[redacted]

Notes

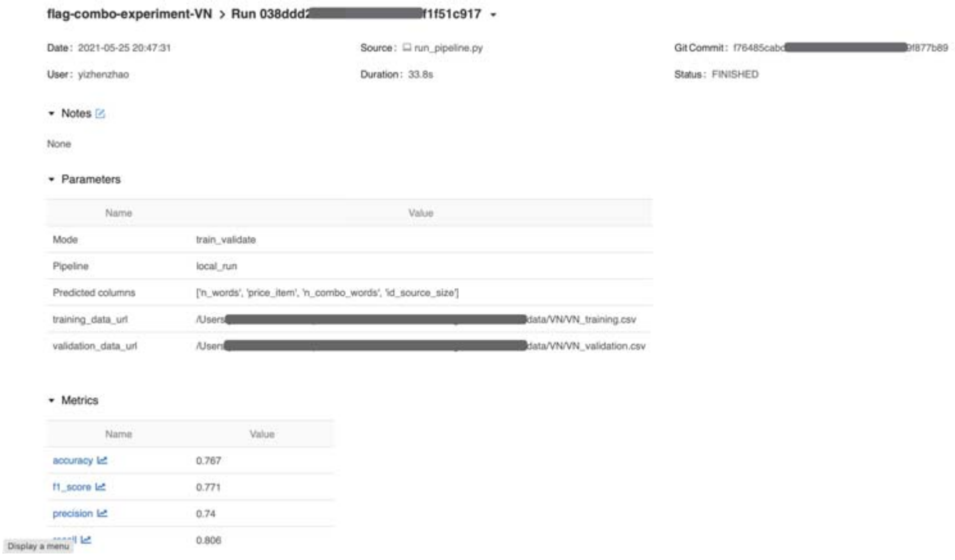
None

Search Runs: metrics.rmse < 1 and params.model = "tree" and tags.mlflow.source.type = "LOCAL"   Filter   Search   Clear

Showing 21 matching runs   Compare   Delete   Download CSV   Columns

						Parameters			Metrics		
Start Time	Run Name	User	Source	Version	Models	Mode	Pipeline	Predicted	accuracy	f1_score	precision
2021-05-28 17:4	-	yizhe...	run_pi	2f2b06	-	validate	local_...	-	-	-	-
2021-05-28 17:4	-	yizhe...	run_pi	2f2b06	-	train_...	local_...	-	-	-	-
2021-05-25 20:4	-	yizhe...	run_pi	f76485	sklearn	train_...	local_...	['n_w...	0.767	0.771	0.74
2021-05-25 20:4	-	yizhe...	run_pi	f76485	-	validate	local_...	-	0.789	0.79	0.768
2021-05-25 20:1	-	yizhe...	run_pi	f76485	-	validate	local_...	-	0.789	0.79	0.768
2021-05-25 19:5	-	yizhe...	run_pi	f76485	-	validate	local_...	-	0.782	0.783	0.761
2021-05-25 19:0	-	yizhe...	run_pi	f76485	-	validate	local_...	-	0.782	0.783	0.761
2021-05-25 19:0	-	yizhe...	run_pi	f76485	-	train_...	local_...	-	-	-	-
2021-05-25 18:2	-	yizhe...	run_pi	f76485	sklearn	train_...	local_...	['n_w...	0.782	0.783	0.761

Fig. 5 MLflow Tracking Interface (with different experiments)



flag-combo-experiment-VN > Run 038ddd[redacted]f1f51c917

Date: 2021-05-25 20:47:31   Source: run_pipeline.py   Git Commit: f76485cabd[redacted]9f877b89

User: yizhenzhao   Duration: 33.8s   Status: FINISHED

Notes

None

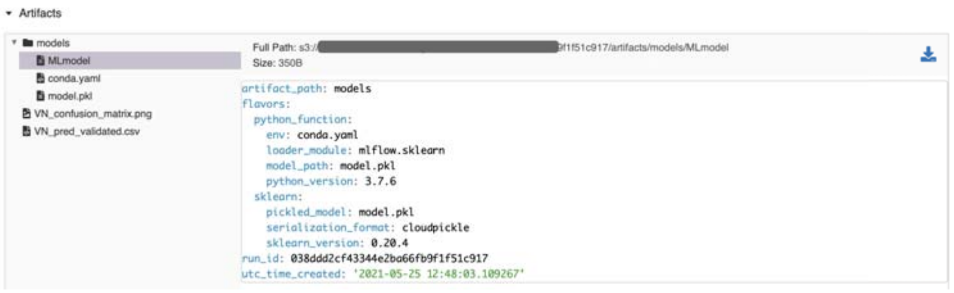
Parameters

Name	Value
Mode	train_validate
Pipeline	local_run
Predicted columns	['n_words', 'price_item', 'n_combo_words', 'id_source_size']
training_data_url	/Users[redacted]data/VN/VN_training.csv
validation_data_url	/Users[redacted]data/VN/VN_validation.csv

Metrics

Name	Value
accuracy	0.767
f1_score	0.771
precision	0.74
[illegible]ll	0.806

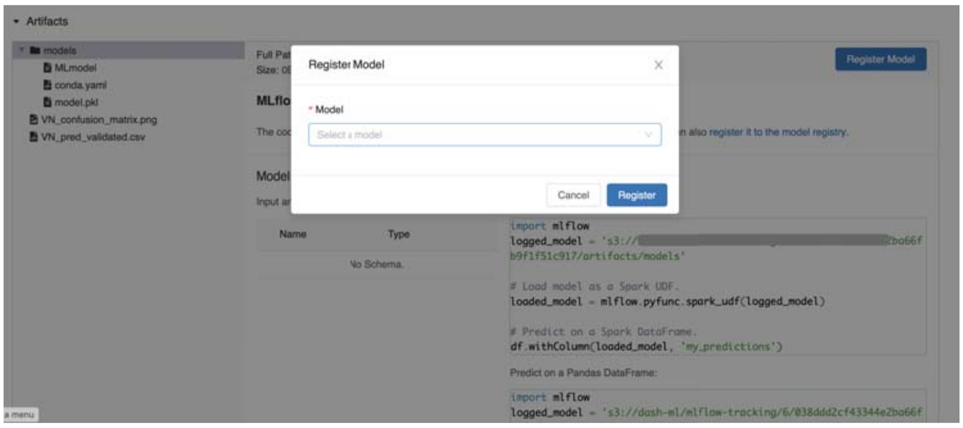
Fig. 6 One experiment details in MLflow Tracking



Full Path: s3://[redacted]9f1f51c917/artifacts/models/MLmodel
Size: 350B

```
artifact_path: models
flavors:
  python_function:
    env: conda.yaml
    loader_module: mlflow.sklearn
    model_path: model.pkl
    python_version: 3.7.6
  sklearn:
    pickled_model: model.pkl
    serialization_format: cloudpickle
    sklearn_version: 0.20.4
run_id: 038ddd2cf43344e2ba66fb9f1f51c917
utc_time_created: '2021-05-25 12:48:03.109267'
```

Fig. 7 MLflow Model, model has been logged into MLflow Tracking. It describes details of the ML model, prerequisite of Model Registry



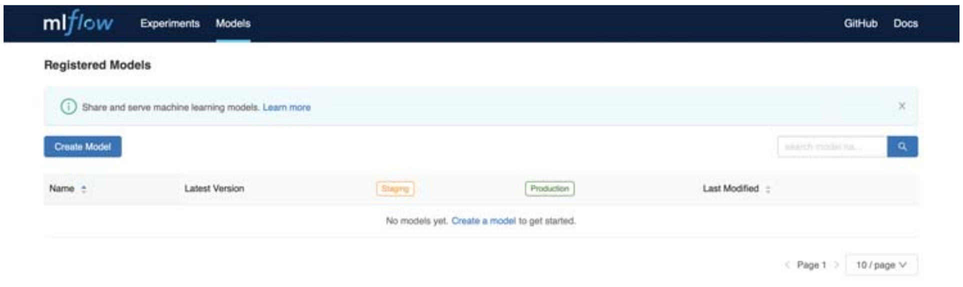
```
import mlflow
logged_model = 's3://[redacted]2ba66f
b9f1f51c917/artifacts/models'

# Load model as a Spark UDF.
loaded_model = mlflow.pyfunc.spark_udf(logged_model)

# Predict on a Spark DataFrame.
df.withColumn(loaded_model, 'my_predictions')
```

```
import mlflow
logged_model = 's3://dash-ml/mlflow-tracking/6/038ddd2cf43344e2ba66f
```

Fig. 8 Model Registry, registered model will be kept in a centralized place. Shown in next Fig.



Registered Models

Share and serve machine learning models. Learn more

Name	Latest Version	Staging	Production	Last Modified

No models yet. Create a model to get started.

Fig. 9 Centralized place for registered models, with stages of each model (e.g.staging, production).

References

[1] Sculley, D. and Holt, Gary and Golovin, Daniel and Davydov, Eugene and Phillips, Todd and Ebner, Dietmar and Chaudhary, Vinay and Young, Michael and Crespo, Jean-Francois and Dennison, Dan, *Hidden Technical Debt in Machine Learning Systems*. Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2, NIPS'15, page 2503–2511, Cambridge, MA, USA, MIT Press, 2015.

[2] Dashmote https://dashmote.com/

[3] Yizhen Zhao, *Machine Learning in Production: A Literature Review*. https://staff.fnwi.uva.nl/a.s.z.belloum/LiteratureStudies/Reports/2021-LiteratureStudy-report-Yizhen.pdf

[4] Adarsh Shah, *Challenges Deploying Machine Learning Models to Production*. https://towardsdatascience.com/challenges-deploying-machine-learning-models-to-production-ded3f9009cb3

[5] Luigi, *5 Challenges to Running Machine Learning Systems in Production*. https://mlinproduction.com/5-challenges-to-ml-in-production-solve-them-with-aws-sagemaker/

[6] Paleyes, Andrei and Urma, Raoul-Gabriel and Lawrence, Neil D. *Challenges in Deploying Machine Learning: a Survey of Case Studies*. arXiv e-prints, page arXiv:2011.09926, 2020.

[7] Git https://git-scm.com

[8] Anant Bhardwaj and Souvik Bhattacherjee and Amit Chavan and Amol Deshpande and Aaron J. Elmore and Samuel Madden and Aditya G. Parameswaran *DataHub: Collaborative Data Science & Dataset Version Management at Scale*, 2014.

[9] Datahub https://datahub.io/

[10] Vimarsh Karbhari, *MLOps: Data Science Version Control*. https://medium.com/acing-ai/ml-ops-data-science-version-control-5935c49d1b76

[11] Pachyderm https://www.pachyderm.com/

[12] AWS Sagemaker Ground Truth https://aws.amazon.com/sagemaker/groundtruth/?nc1=h_ls

[13] AWS Sagemakerweb https://aws.amazon.com/sagemaker/

[14] Azure https://azure.microsoft.com/en-us/

[15] Azure, *Deploy machine learning models to Azure*. https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-and-where?tabs=azcli

[16] MLflow https://mlflow.org

[17] MLflow sagemaker https://www.mlflow.org/docs/latest/python_api/mlflow.sagemaker.html\#module-mlflow.sagemaker

[18] Kyle Gallatin, *Deploying Models to Production with Mlflow and Amazon Sagemaker*. https://towardsdatascience.com/deploying-models-to-production-with-mlflow-and-amazon-sagemaker-d21f67909198

[19] Emmanuel Raj, *Edge MLOps framework for AIoT applications, Continuous delivery for AIoT, Big Data and 5G applications*, 2020.

[20] Azure Machine Learning https://azure.microsoft.com/en-us/services/machine-learning/

[21] DevOps https://azure.microsoft.com/en-us/services/devops/

[22] Pölöskei, István, *MLOps approach in the cloud-native data pipeline design*. Acta Technica Jaurinensis, 2020.

[23] Yizhen Zhao, *MLOps Scale ML in an Industrial Setting*. https://staff.fnwi.uva.nl/a.s.z.belloum/MSctheses/MScthesis_Yizhen_Zhao.pdf

[24] Yizhen Zhao, *MLOps and data versioning in machine learning project*. https://staff.fnwi.uva.nl/a.s.z.belloum/LiteratureStudies/Reports/2020-Internship_report-Yizhen.pdf

[25] Yizhen Zhao, *MLOps: Data versioning with DVC — Part I*. https://yizhenzhao.medium.com/mlops-data-versioning-with-dvc-part-\%E2\%85\%B0-8b3221df8592

[26] Ubereats https://www.ubereats.com/nl-en

[27] DVC https://dvc.org/

[28] Jenkins https://www.jenkins.io/

[29] DVC File https://dvc.org/doc/user-guide/project-structure/dvc-files\#dvc-files

[30] Airflow https://airflow.apache.org/

[31] Git Flow https://guides.github.com/introduction/flow/

[32] DVC YAML File https://dvc.org/doc/user-guide/project-structure/pipelines-files

[33] DVC LOCK File https://dvc.org/doc/user-guide/project-structure/pipelines-files\#dvclock-file

[34] Sagemaker Batch Transform https://docs.aws.amazon.com/sagemaker/latest/dg/batch-transform.html

[35] Yizhen Zhao, *MLOps: Deploy custom model with AWS Sagemaker batch transform — Part II*. https://yizhenzhao.medium.com/mlops-deploy-custom-model-with-aws-sagemaker-batch-transform-part-\%E2\%85\%B1-54263ec711ce

[36] Sagemaker Price https://aws.amazon.com/sagemaker/pricing/

[37] MLflow, *How Runs and Artifacts are RecordedHow Runs and Artifacts are Recorded*. https://mlflow.org/docs/latest/tracking.html\#how-runs-and-artifacts-are-recorded
[38] AWS EC2 Target Group https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-target-groups.html
[39] AWS, *Train a Model with Amazon SageMaker*. https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-training.html
[40] Azure, *Machine Learning Operations maturity model*. https://docs.microsoft.com/en-us/azure/architecture/example-scenario/mlops/mlops-maturity-model