A MACHINE LEARNING PIPELINE IMPLEMENTATION USING MLOPS AND GITOPS PRINCIPLES

Nataša Radaković [1,2] [ORCID 0009-0004-0221-5037], *Ivana Šenk* [2] [ORCID 0000-0002-8296-7885], *Nina Romanić* [1]
[1] Productdock d.o.o, Novi Sad, Serbia
[2] University of Novi Sad, Faculty of Technical Sciences, Serbia

Abstract: Automating machine learning model development and deployment improves speed, efficiency, reproducibility, scalability, reliability, and cost efficiency. It enables organizations to iterate quickly, scale effectively, reduce errors, and stay competitive in the rapidly evolving field of machine learning. The aim of machine learning operations (MLOps) is to automate the entire lifecycle of the model, from development to deployment and management in production. The paradigm of MLOps refers to the practice of applying DevOps principles and practices to the machine learning lifecycle. It combines various disciplines such as machine learning, software engineering, data engineering, and operations. Different MLOps tools are available. Each tool has its own strengths and features and its usage depends on specific requirements, infrastructure, and preferences. Organizations can combine MLOps and GitOps concepts to achieve more efficient and controlled deployments and management of ML models. In this paper we describe the machine learning pipeline that we created using available tools and following MLOps and GitOps practices.

Key words: machine learning, MLOps, GitOps, CI/CD, operations, workflow orchestration

1. INTRODUCTION

Machine learning has become an ubiquitous term in the operations sector, buzzing with significance. Companies are actively integrating intelligent systems into their legacy infrastructure. Machine learning (ML) teams typically comprise a diverse set of professionals, including data scientists, software engineers, and machine learning engineers. MLOps enhances team collaboration by bridging the gap between development and operations, thereby improving the reliability of product development, deployment, and management (Kreuzberger et al. 2023, Testi et al. 2022). MLOps (Machine Learning Operations) is a paradigm that emphasizes the operational aspects of the machine learning lifecycle, encompassing implementation, deployment, monitoring, and management of machine learning products. By leveraging Continuous Integration/Continuous Delivery (CI/CD) practices, MLOps enables the automation of the deployment process, facilitating efficient software development (Subramanya et al. 2022).

GitOps practices and existing ML tools enhance the development and deployment of ML models, making it more efficient and straightforward. GitOps is an infrastructure and application deployment methodology that leverages Git workflows and CI/CD. All configuration files are treated as code, known as infrastructure as code (IaC), providing declarative descriptions of the desired system state. In GitOps, these files serve as the single source of truth for infrastructure and application deployments, enabling automated processes to reconcile the actual state with the desired state. Similar to how application source code generates consistent application binaries during builds, GitOps ensures that each deployment consistently generates the same infrastructure environment (Alonso et al. 2023, Yuen et al. 2021).

Various ML tools and models are available in both practice and the market. However, it's worth noting that not all ML tools in the market align seamlessly with every business's requirements. Research papers often provide comprehensive assessments of these tools and methodologies, outlining their advantages and disadvantages (Schlegel and Sattler (2022)). In this particular study, our aim was to explore and analyze the processes and open source tools that can be effectively applied to a machine learning problem following GitOps practices and providing infrastructure as a code (IaC). We created end-to-end MLOps workflow by applying the knowledge we acquired. When incorporating machine learning models into an existing legacy system, it is crucial to carefully select tools that seamlessly integrate with the system and align with the capabilities of the existing teams. In this paper we are describing practices and tools that can be applied to a legacy system with web application deployed on a cloud kubernetes cluster. We described an end-to-end MLOps architecture and workflow that takes into account software engineering practices, tailored specifically for this system.

2. OVERVIEW OF UTILIZED TOOLS

2.1 Model development and experimentation

Data scientist usually use Jupiter Notebook for writing the code. Notebooks are easy to start and use. At the beginning when you are developing proof of concept (POC) this can be enough. Once you have the POC it is time for experimentation phase. In this phase models, hyperparameters and algorithms can change in order to get the best solution. It starts to be hard to keep track of all those models, and results. It is important for results to be reproducible, and to know what model, data, and hyperparameters gave specific results. Also, in real life, after some time data sets can change. In these cases notebooks are not enough. Based on various tool reviews and considering the presence of existing legacy systems, we have opted to utilize Iterative AI Tools consisting of data version control (DVC), Continuous Machine Learning (CML) and Git Tag Ops (GTO), that enable us to effectively monitor and replicate our experiments, and also track ML model lifecycle.

Data versioning is a vital role in MLOps and it is challenging to handle when the dataset is large. Usually Git is used for versioning the code, but it is not meant to be used for versioning large data. Due to its limited upload capacity, this option is not suitable when dealing with large data. However, in such cases, DVC can be a valuable solution. DVC is an open-source platform-independent versioning system for ML applications. It helps machine learning teams manage large datasets, make projects reproducible, and collaborate better (Hewage and Meedeniya 2022). DVC enables the storage of files in a remote storage location, overcoming the limitations associated with handling substantial amounts of data. Metadata files with information about files are stored in git, but actual data is stored in DVC. That metadata file is used for data retrieval from DVC.

During the development of a model, we typically follow a series of steps, including data preprocessing, training, and evaluation. In DVC, these steps are referred to as stages. Each stage consists of a command and its dependencies, and it allows for the addition of outputs. The outputs generated by one stage can serve as inputs to subsequent stages. Stages are described in a .yaml file, which provides a clear overview of what is expected from each stage. DVC determines the order in which the stages should be executed and can cache the results of stages that haven't changed. This caching mechanism helps save time by running only the stages that have been modified. However, if the requirements file is changed, all stages will be executed. Moreover, if the dependencies for a particular stage are not met, that stage will not be executed. Outputs produced by stages can be versioned using DVC or git.

These stages, with input and output, can be viewed as experiments within the GitOps approach. In this context, experiments correspond to Git commits, eliminating the need for a separate database. Moreover, we can create distinct branches for individual experiments, allowing for parallel exploration. An editor extension further enhances the process, providing a user-friendly interface to visualize and track experiments. With the dashboard, there's no requirement to save all experiments. Instead, we can selectively choose a few and generate Git commits, discarding the rest. This lightweight, command-line tool can be seamlessly integrated into DevOps workflows, facilitating efficient experimentation.

In addition to DVC, the iterative enterprise encompasses Continuous Machine Learning (CML), which plays a vital role in enabling CI/CD for ML projects. CML relies on GitLab or GitHub actions to effectively manage ML experiments, track modifications, and automatically generate reports containing essential metrics and plots within each Git pull request. When a team of people is developing a ML model, they typically create a pull request and also include plots and metrics in the comments. Frequent iterations between data scientists are often necessary, as they need to reach a mutual understanding of the meaning and context behind these screenshots. With CML reports are automatically generated and displayed in the PR comments. In this way the entire team gains awareness of the conditions under which these plots and metrics appeared. This simplifies collaboration and pull request reviews, eliminating the need for additional screenshots, enhancing the efficiency of the development workflow.

By leveraging DVC and CML, we have successfully addressed the development and experimentation aspects of the model within a GitOps paradigm. Moving forward in the end-to-end pipeline, the next crucial component to consider is a model registry. A model registry serves as a valuable tool for cataloging ML models and their respective versions. It enables seamless discovery, testing, sharing, deployment, and auditing of models from your data science projects (GTO 2023).

To fulfill our model registry requirements, we will utilize the GTO (Git Tag Ops) tool developed by Iterative. As a product from the same company, GTO seamlessly integrates with DVC and CML, ensuring compatibility across the entire workflow. GTO follows the GitOps approach, eliminating the need for separate databases or servers, and solely relies on the DVC and Git repository. Additionally, GTO provides support for model promotion to specific stages, allowing the model to be effectively utilized within designated environments. By assigning a specific stage and utilizing the model registry, we establish a structured approach to manage and track the progression of the ML model throughout its lifecycle. This allows for easy version control, collaborative development, and seamless integration with the deployment pipeline for subsequent stages, such as model deployment and serving.

2.2 Serving a model

To serve our model, we have two options: utilizing a REST API for real-time predictions or using a scheduler for batch processing. In the context of our ML model within a legacy system, we have chosen to adopt real-time predictions through a REST API. Our approach involves starting with a Flask application, which serves as the foundation for our system. To create a robust web server, we connect uWSGI to Flask. uWSGI acts as an application server, facilitating communication with Flask using the uwsgi protocol. To enhance connection handling reliability, we incorporate Nginx as a reverse proxy. Nginx acts as an intermediary between client requests and our Flask application, ensuring efficient and secure communication. By combining Flask, uWSGI, and Nginx, we establish a comprehensive architecture for serving our model and providing real-time predictions via a REST API. (Relan 2019)

2.3 Deployment and Scalability with Containerization

There are third-party tools available for containerizing ML models. These tools are typically designed to be user-friendly and offer support for various platforms. However, one drawback is that they are often presented as black boxes, making them difficult to debug. Additionally, there is uncertainty regarding their compatibility with air-gapped networks. Therefore, we have chosen to create our own Dockerfile to have more control and flexibility in these aspects. To containerize the ML model (Karamitsos 2020) using Docker, we will follow a two-stage approach defined in the Dockerfile. In the first stage, we will download the model from the GTO registry using DVC. The downloaded model will then be stored in a specified path.

Moving on to the second stage in the Dockerfile, we will install all the necessary dependencies required for serving the ML model. This ensures that the container has all the required software components to run the model successfully. Additionally, to enhance security, we will configure the container to run as a non-root user, mitigating potential risks.

Finally, the Dockerfile will enable the container to serve the ML model, allowing it to handle incoming requests and provide predictions or insights based on the input data. Our ML model will not be directly exposed to the public network. Instead, we will have a backend service that will communicate with the model via its API.

By leveraging Docker technology and following this approach, we can encapsulate the ML model along with its dependencies and security measures within a self-contained and portable container. This containerization process greatly simplifies the deployment and scaling of the model across various environments, ensuring consistent performance and ease of management (Filippou 2023).

3. CONTINUOUS INTEGRATION AND WORKFLOW EXAMPLE

Figure 1 shows the MLOps pipeline for CI/CD. During the experimentation phase, we will create different branches for model testing and hyperparameter tuning. These branches correspond to the development environment, which is meant for separate ML model testing. Training input data, test data and model are stored on a remote storage, using DVC. Whenever we push something to these branches, GitHub actions are triggered, and a CML report is printed in the comments. We will also utilize GitHub actions to register and push the model in the GTO registry. If a model with the same name already exists, its version will be incremented, and the tag will be pushed to the Git repository.

In our Git repository, we have three additional branches: "main", "staging" and "production." The "main" branch corresponds to the development environment, while the "staging" and the "production" branch refers to the staging and production environment, consequently. In development environment, the model is seamlessly integrated with the existing legacy system and thoroughly tested together. This integration and testing phase ensures that both the model and the code function harmoniously within the larger system. Once the model and code pass all the necessary tests in the development environment it is deployed to staging environment, following the process shown in Figure 1. A staging stage serves as a checkpoint to ensure the model is thoroughly evaluated before moving it to production. When the model successfully passes all the required tests in the staging environment, the staging branch is prepared for merging into the production branch.

When we are satisfied with an experiment and the model in a specific (experimental, development, or staging) branch, and when we want to deploy this model to the specific environment (development, staging, or production), we create a pull request (PR) to merge this branch with the branch that represents the environment where we want to deploy the model. We can only merge the experimental branch into the development branch, the development branch into the staging branch, and the staging branch into the production branch. This approach ensures that the model successfully traverses all the required environments and passes all necessary tests. It is expected that, after merging and utilizing CI/CD processes, this code and the model will automatically end up in the development and staging environment.

When a pull request (PR) is created, a GitHub action will be triggered. In the first step of the action, the corresponding model for the branch being merged will be obtained. Leveraging DVC, a simple 'dvc pull' command is sufficient to download the required model based on the metadata stored in the Git repository for that specific branch. Once obtained, the model will be assigned a 'dev', 'staging' or 'production' stage, depending on the branch to which the code is being merge, indicating that it is ready for further testing or validation. The model version, along with its associated artifacts, will then be pushed to the model registry. Moving on to the second stage of the action, the model will be packed into a Docker image. The Docker image will be built using the instructions provided in the Dockerfile. To ensure proper versioning, the image will be tagged using the git commit hash, which serves as a convenient way to associate the image with the specific code corresponding to it. Additionally, the image will be pushed to the container registry, making it accessible for deployment and utilization. Finally, when GitHub actions are triggered, a CML report is printed in the comments of the PR.

By following this workflow, the GitHub action automates the process of obtaining the required model, building the Docker image, and versioning it appropriately. This streamlined approach simplifies the deployment and management of the ML model, ensuring that the correct model and corresponding code are utilized in each image.

When the model successfully passes all the required tests in the staging environment, the staging branch is prepared for merging into the production branch. A pull request (PR) is created for this purpose. We repeat the GitHub actions for model promotion and pushing to the registry, but this time the model is promoted to the "production" stage. However, it's important to note that the code and model changes are not directly applied to the live environment. The deployment strategy for the production environment can vary depending on the specific problem and requirements. Some strategies may include automatic promotion without customers noticing the deployment, while others may require temporary unavailability of the system during manual trigger-based deployments. Since the deployment strategies for the production environment are highly specific to the problem at hand, it is beyond the scope of this paper to delve into them further. By adhering to these practices, we ensure a controlled and reliable deployment process that incorporates collaboration, thorough testing, and appropriate promotion of the model and code to the production environment.

In our workflow, we prioritize a collaborative and controlled approach for applying changes to the production environment. To ensure quality and consistency, we have specific requirements in place. Firstly, we require at least one PR approval, signifying that the proposed changes have been reviewed and accepted by the team. Additionally, we expect all tests, as defined in the GitHub repository, to pass successfully. To maintain a structured and reliable deployment process, direct pushes to the main, staging or production branches are not allowed.

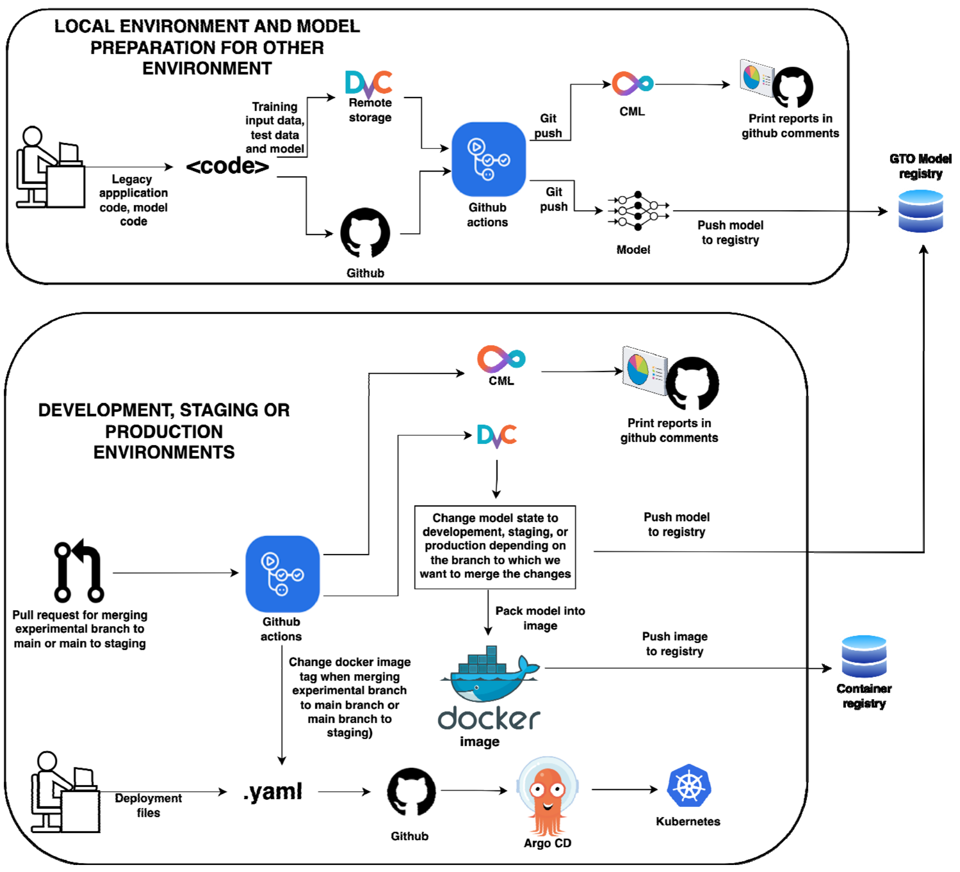
LOCAL ENVIRONMENT AND MODEL PREPARATION FOR OTHER ENVIRONMENT
Legacy application code, model code
<code>
Training input data, test data and model
Remote storage
Github
Github actions
Git push
CML
Print reports in github comments
Git push
Model
Push model to registry
GTO Model registry
DEVELOPMENT, STAGING OR PRODUCTION ENVIRONMENTS
Pull request for merging experimental branch to main or main to staging
Github actions
CML
Print reports in github comments
Change model state to developement, staging, or production depending on the branch to which we want to merge the changes
Push model to registry
Pack model into image
Push image to registry
Container registry
Change docker image tag when merging experimental branch to main branch or main branch to staging)
docker image
Deployment files
.yaml
Github
Argo CD
Kubernetes

Figure 1: ML pipeline for CI/CD

4. DEPLOYING TO KUBERNETES

To maintain the GitOps approach throughout the deployment phase, we have established a separate repository to store all the deployment-related files for ML model and also the existing legacy code. For deploying the model, we will leverage the Argo CD tool. Argo CD is a declarative, GitOps continuous delivery tool designed specifically for Kubernetes environments (Argo 2023). In our deployment setup, Git repositories serve as the source of truth, defining the desired application state. We will have a deployment file that represents the native Kubernetes operator (Kubernetes 2023). This file will specify details such as the container image with its tag, the command to be executed, the required resources, and health checks. Additionally, we will have a separate file for the service definition that specifies the deployments. We will have all these files for every environment. So we will have three sets of these files. To utilize Argo CD, it will be installed in the Kubernetes cluster. Argo CD provides a user-friendly UI that simplifies tracking and monitoring of the deployed applications across various environments. In the YAML files we will provide information about the Git repository and the branch we want to deploy from. Argo CD will then compare the provided YAML files with the current state of the cluster, identifying any discrepancies. Argo CD looks at the provided yaml files and to the cluster and tells if it is out of sync. When you initiate a sync operation in Argo CD, it will deploy any differences detected between the desired state defined in the Git repository and the current state of the cluster. It offers an automated sync policy, which means that every time a commit and push occurs in the Git repository with changes to the manifests in the tracking Git repository, Argo CD will automatically perform the deployment.
In Figure 1, we can see how Argo CD fits in the whole pipeline. In addition to pushing the versioned and staged model to the model registry and pushing the Docker image to the registry when merging the experimental branch to the development branch or development to staging, GitHub Actions will include a third step. This step will modify the repository's deployment files by updating the Docker image tag for the specific environment. For example, if we are merging the experimental branch to the main branch, it will modify the deployment file for the development stage, allowing us to observe the new container in the development stage. Similarly, it will update the staging deployment file when merging the main branch to staging. This process enables Argo CD, which is set to automatic sync policy, to compare the

newly provided YAML file and identify the changed Docker tag. As a result, Argo CD will automatically deploy the new model and image to the corresponding environment. Additionally, Argo CD will update the environments every time the developer makes some changes to the deployment metadata files.
By leveraging Argo CD's GitOps approach, we ensure a streamlined deployment process, where the desired state is defined in Git repositories and seamlessly synchronized with the Kubernetes cluster. This setup enables easy tracking, automated deployments, and ensures consistency across different environments.

5. CONCLUSIONS

In this paper, we described the machine learning pipeline that can be applied to the legacy software system already deployed in the cloud. Using this end-to-end pipeline, the process can be fully automated using open-source GitOps tools. These tools have helped us automate the entire ML model development lifecycle, from training to deployment. The proposed workflow ensures that ML models are always up to date and in sync with the latest changes in the code. It enables us to version and track not only the model code but also the data, hyperparameters, training configurations, and experiments. By maintaining a history of experiments and their associated results, we can make more informed decisions about model improvements and optimizations. By incorporating GitHub Actions, following GitOps principles, and utilizing GitOps tools, you can streamline the development, deployment, and maintenance of ML models, fostering collaboration, reproducibility, and overall quality. In our future work, we will attempt to implement the proposed pipeline using the recommended tools. Additionally, we will explore deployments strategy in order to apply it to the production environment. Finally, we will explore methods for model retraining and monitoring.

6. REFERENCES

Alonso, J., Piliszek, R., Cankar, M. (2023) Embracing IaC Through the DevSecOps Philosophy. IEEE software |published by the IEEE computer society. Available from: doi: 10.1109/MS.2022.3212194.

Argo (2023) Available from: https://argo-cd.readthedocs.io/en/stable/ [Accessed 25th june 2023].

Filippou, K., Aifantis, G., Papakostas, G.A., Tsekouras, G.E. (2023) Structure Learning and Hyperparameter Optimization Using an Automated Machine Learning (AutoML) Pipeline. Information 2023, 14,232. Available from: doi: 10.3390/ info14040232.

GTO (2023) Avaliable from: https://mlem.ai/doc/gto [Accessed 25th june 2023].

Hewage, N., Meedeniya, D. (2022) Machine Learning Operations: A Survey on MLOps Tool Support. pp. 1-12, Arxiv.2202.10169. Available from: doi: 10.48550/arXiv.2202.10169.

Karamitsos, I., Albarhami, S. and Apostolopoulos, C. (2020) Applying DevOps Practices of Continuous Automation for Machine Learning. Information 2020, 11, 363. Available from:doi: 10.3390/info11070363.

Kreuzberger, D., Kühl, N. And Hirschl, S. (2023) Machine Learning Operations (MLOps): Overview, Definition, and Architecture. IEEE Access. Available from: doi: 10.1109/ACCESS.2023.3262138.

Kuberenetes (2023) Available from: https://kubernetes.io/ [Accessed 25th june 2023].

Relan, K. (2019). Deploying Flask Applications. In: Building REST APIs with Flask. Apress, Berkeley, CA. Avaliable from doi: 10.1007/978-1-4842-5022-8_6.

Schlegel, M., Sattler, K. (2022) Management of Machine Learning Lifecycle Artifacts: A Survey. SIGMOD Record, December 2022 (Vol. 51, No. 4).

Subramanya, R., Sierla, S. and Vyatkin, V. (2022) From DevOps to MLOps: Overview and Application to Electricity Market Forecasting. Electricity Market Forecasting. Appl. Sci.2022,12,9851. Available from: doi: 10.3390/app12199851.

Testi, M., Ballabio, M., Frontoni, E., Iannello, G., Moccia, S., Soda, P. And Vessio, G. (2022) MLOps: A Taxonomy and a Methodology. IEEE Access. Available from: doi: 10.1109/ACCESS.2022.3181730.

Yuen, B., Matyushentsev, A., Ekenstam, T. and Suen, J. (2021) GitOps and Kubernetes. Shelter Island, Manning Publications Co.