Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 53-5 (2020) 704–709

Building A Platform for Machine Learning Operations from Open Source Frameworks

Yan Liu [*] Zhijing Ling [*] Boyu Huo [*] Boqian Wang [*]
Tianen Chen [*] Esma Mouine [*]

[*] *Gina Cody School of Engineering and Computer Science, Concordia University, Montreal QC H3G 1M8, Canada (e-mail: yan.liu@concordia.ca)*

Abstract
Machine Learning Operations (MLOps) aim to establish a set of practices that put tools, pipelines, and processes to build fast time-to-value machine learning development projects. The lifecycle of machine learning project development encompasses a set of roles, stacks of software frameworks and multiple types of computing resources. Such complexity makes MLOps support usually bundled with commercial cloud platforms that is referred as vendor lock. In this paper, we provide an alternative solution that devises a MLOps platform with open source frameworks on any virtual resources. Our MLOps approach is driven by the development roles of machine learning models. The tool chain of our MLOps connects to the typical CI/CD workflow of machine learning applications. We demonstrate a working example of training and deploying a machine learning model for the application of detecting software repository code vulnerability.

Keywords: Machine Learning, DevOps, Software Architecture, Open Source.

1. INTRODUCTION

The concept of Machine Learning Operations (MLOps) originates from DevOps that aims to establish a set of practices that put tools, pipelines, and processes to build machine learning development in a timely manner and cost efficiently. One major difference between DevOps and MLOps relies on the fact that software codifies actions. Different scopes of test are placed along the development of software in the CI/CD lifecycle. On the other hand, machine learning models are constructed entirely. Models are then trained and tuned by hyper-parameters and various datasets.

Industrial practices observe that *an enterprise can take six to 18 months to deploy a single machine learning model to production*. This indicates requirements may change on both machine learning models and machine learning applications. Hence the cycle of MLOps should connect to the DevOps cycle to ensure the CI/CD workflow of a machine learning application. DevOps is in action after a machine learning model is deployed on the hosting application. Till then, MLOps spans over the phases of model coding, training, inspection, reviewing and testing as inference in iteration. Finally MLOps supports the model deployment to the hosting application. The bridge to connects MLOps and DevOps cycles are the data flows and model flows passing from different roles of data scientists, machine learning project managers, and application developers.

Cloud service platforms are bundled with their cloud infrastructures for computing resources of GPUs, CPUs and storage. For example, model development in Amazon SageMaker has automation support for training, experiments, hyper-parameter optimization and deployment. Both SageMaker and Azure Machine Learning have provision of models from pre-built algorithms. Google's Colab provides a Jupyter notebook for any model construction but less support for model deployment. Comparison of public cloud platforms indicates different roles involved in machine learning development are not explicitly addressed by these platforms.

This leads to the core research question in this paper as ***how model flows and data flows handled by specific roles can converge to the CI/CD workflow of machine learning application at which the machine learning model is deployed?*** We further compose this question into threefold as follows:

(1) What are the cooperation activities among roles at phases of machine learning development?
(2) How to automate model deployment for end machine learning applications?
(3) How provision of data and computing resources is designed by the system architecture?

Our research method is driven by the cooperation activities in alignment with the machine learning development phases. These activities demand provision of computing and storage resources. We build up a platform called OCDL that provisions virtual computing instances of GPUs and CPUs. A middleware layer is built for data acquisition from both public repositories and from proprietary computing nodes. Using the middleware APIs, data are accessed at the memory speed for data persistent on HDFS nodes. On our OCDL platform, models are coded and trained. Moreover, the operations supported by OCDL ensure further model oriented tasks of role-based model

10.1016/j.ifacol.2021.04.161

review, inspection, inference testing, the approval process and automated model deployment.

To validate our OCDL platform, we build a machine learning model that learns code vulnerability signature from 17 Github public projects. The model is assessed before deployed to a code vulnerability detection application. We run this application on our own OCDL repository and identify 15 spots of vulnerable codes.

The structure of this paper is organized as follows. Section 2 presents the related work. In Section 3, we present the methodology. Section 4 discusses the system architecture of the OCDL. We demonstrate our case study in Section 5. Section 6 concludes our paper.

2. RELATED WORK

We discuss the state-of-the-art work related to CI/CD of machine learning models and operations in two aspects: 1) adoption of DevOps for machine learning; 2) frameworks and platforms for operating data, models and computing resources.

2.1 DevOps for Machine Learning Applications

DevOps has been applied with agile and lean software development techniques and tools. The adoption of DevOps to manage data analytics and machine learning software has been reported in various domains. The work in Erich et al. (2017) described the results of an exploratory interview-based study involving six organizations of various sizes that are active in various industries. They observed that all organizations were positive about their experiences and only minor problems were encountered while adopting DevOps. Rothenhaus et al. (2018) proposed a reference architecture for fleet administration software CI/CD in US Navy. Comparing to COTS machinle learning platforms, this reference architecture captures the release governance process that is an essential for approval before the deployment phase. Specifically, a software architecture is proposed by Ciucu et al. (2019) for managing virtual resources and data structure for machine learning applications. AWS IoT Greengress provides a user interface to deploy machine learning models by means of AWS Lambda functions. This enables machine learning model becomes a service running on Docker images. A practice has been demonstrated by Krishnamurthi et al. (2019) on deployment of machine learning models to IoT devices.

2.2 Frameworks and Platforms

Kuaa is a workflow based framework for designing, deploying and executing machine learning experiments proposed by de Oliveira Werneck et al. (2018). It consists of built in tasks such as feature descriptors, data normalizers, classifiers and data fusions. Tasks are components to form a workflow. One limitation is to explore new types of models. In addition, the computing resources are also essential to be integrated with the automated machine learning workflows. To address this technology challenge, GPU and CPU clusters have been built up in enterprise data centers presented by Zou et al. (2014).

Miao et al. (2017) addressed the reuse of models in the lifecycle management of deep learning. The solution proposes (1) a model version system, (2) model configuration language and (3) a model sharing system to publish, discover and reuse models. López García et al. (2020) suggested models serving as a service, and proposed a distributed architecture to provide a set of tools and cloud services ranging from the models creation, training, validation and testing to the models serving as a service, sharing and publication.

2.3 Computing and Storage Resource Management

Computing resources and storage resources are two pillars of machine learning development. Any operations for machine learning development should be connected to computing and storage resource operations. The process level virtualization adopts container-based Docker images. The virtual node clustering solution adaopts Kubernetes to build the base for a variety of technical implementation in the works of Medel et al. (2016), Surya and Imam Kistijantoro (2019), Wang et al. (2020), and Chang and Zha (2018). To further scale the cluster of virtual nodes, a memory based data access framework Alluxio developed by Li (2018) is introduced over virtual clusters managed by Kubernetes.

2.4 Summary

Existing research work extends the practices of DevOps to machine learning development by addressing the machine learning lifecycle, model sharing, publishing and reuse. The solutions of above works need to integrate with virtual and scalable computing clusters. Our work considers the lifecycle of machine learning development driven by roles and their collaborations. We explicitly address the model releasing process and the model's connection to the CI/CD workflows of building machine learning applications. Comparing with existing works, we build up operations and the run-time platform from source frameworks rather than adding extensions to existing DevOps. Our MLOps process and the run-time environment are supported by a layer of virtual resource management. The advantage of our work is it is not specific to a certain domain so that any machine learning model can be developed following MLOps.

3. THE METHODOLOGY

Our methodology follows the roles and their activities during the machine learning development that sets the context of each role. The context is then supported by the model operations and virtual resource access, and data management.

3.1 MLOps Phases

In this paper, we defined 4 phases of MLOps that are consistent with a machine learning's lifecycle : 1) *model programming*, 2) *model training*, 3) *model assessment* and 4) *model releasing*. In the model programming phase, OCDL provides functions that include 1) an online IDE for model coding; 2) a resuable code template; 3) data access

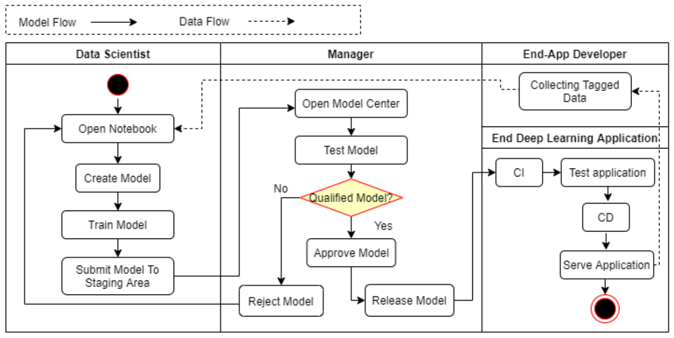


Figure 1. Cooperative activities among different roles

utilities to HDFS or online repositories; and 4) choice of GPU/CPU computing instances.

During the model training phase, OCDL encapsulates the cluster resource provision and scheduling managed by Kubernetes. Such a cluster is heterogeneous. The cluster nodes can be private GPU nodes or pay-as-you-go instances from different public cloud infrastructures. Models are trained, revised and fine-tuned in iteration.

After the model is tested by the model developer, it is then submitted to the next phase for model assessment. This process is akin to component test and system test in software development. Model evaluation and assessment may use different datasets. Model assessment checks how accurate the inference of the machine learning model is. Therefore, model assessment also requires the same workflow environment as model development to access data, launch computing resources and run models.

Finally, the model is approved for releasing. The releasing has two forms. One form is pushing the model to a repository or to the hosting application which subscribes to the model releasing topic. The other form is automatically launching the model as a service. The invocation of the model service is through REST API calls.

3.2 Roles and Cooperation Context

Each phase is operated by roles. Roles cooperate across phases. Figure 1 shows the cooperative activities among different roles.

The data scientist context A data scientist is responsible for creating and training the models. Their work is mostly focused on the first two phases of the model lifecycle (model building and model training). A data scientist may share data and models with other data scientists, data engineers and project managers. The OCDL design provides three working spaces to cover a data scientist's context, namely personal working space (private), project working space (public) and staging space (authorized only). With these separate working spaces, data scientists are able to share their data and models in the public area or isolate their models in progress in a private working space.

The manager context A manager performs the assessment for all models before the model releasing. These models are submitted by data scientists after their private test in their personal working space. The staging space supports the manager's operation on model testing and evaluation. In this process, the manager downloads the model from the staging space to one's private space to run the model on test datasets. Next, the manager decides if the model should be released from the staging space to machine learning applications. Models can be rolled back from staging space back to the scientist's private space for model revision. The staging space allows a machine learning development team to maintain multiple versions of one model as well as multiple model development at the same time.

The end application developer context The end application developer has the role of consuming models. The deployment of models to the end application is within the DevOps process. The MLOps is responsible for connecting to DevOps tool chains. Hence the end application's context is not within the scope of MLOps. In this paper, we refer the last phase of MLOps as the model releasing from the staging space.

3.3 Enabling Machine Learning Application CI/CD

The connection from the model releasing to model deployment merges to the DevOps process. After the model releasing, the model becomes a soft component to be further encapsulated as applications or services in the DevOps process. In this paper, we define MLOps end at the point of model releasing. Hence, we identify the merge points of MLOps and DevOps when the model is committed to the repository of the machine learning application.

We depict in Figure 2 the explicit process from model commitment in MLOps to follow-up operations in DevOps. In this process, a model that passes the accuracy test is forwarded to the model manager with enclosed testing report. The manager who approves the model is responsible for committing the model to the end application repository. This step indicates the entrance to the CI/CD process of the end ML application.

In DevOps, a model commitment event triggers a series of tests as follows. The portability test runs the model on different operating systems, software versions such as JDK, Python or libraries. The function test conducts API-test that encapsulates the invocation of the model. The QA test validates the wrapped model as a service. Software quality test ensures the latency requirement under a stress test. The model is also under stability test. Failures on each test involve model inspection and review participated by data scientists. Once all the tests are passed, the new model is fully deployed to the machine learning application.

In this paper, we emulate the process of the ML application CI/CD using "black box" approach since the CI/CD environment and process are out of the scope of this paper. After the model commitment, we assume passing all the tests leads to the deployment operations while failing any test will rollback the model to the MLOps process. In the OCDL *Model Center* view (details in Section 4.1, the model remains under the responsibility of the model manager whether to assign the model back to the data scientists for another round of model approval process.

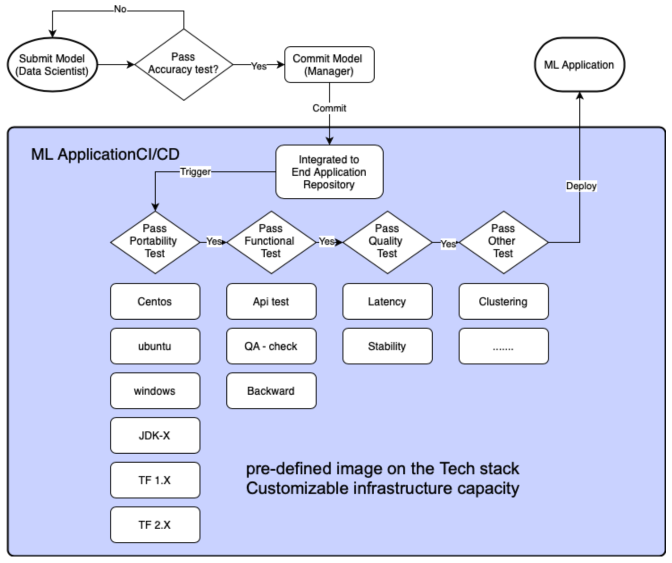


Figure 2. The merge point of MLOps and DevOps upnon deployment of a machine learning model

4. THE SYSTEM ARCHITECTURE

OCDL is an open-source architecture of tool suites that connect models, data, cloud resources and devices for deep learning projects [1] . The architecture consists of three parts, including *Capsule*, *Rover* and *Launcher* in a layered architecture depicted in Figure 3. Capsule is a Web service portal of using the OCDL IDE and runtime. Rover is a resource management framework that provides Capsule the resources on demand. Launcher builds the workflow from model releasing to the end machine learning applications.

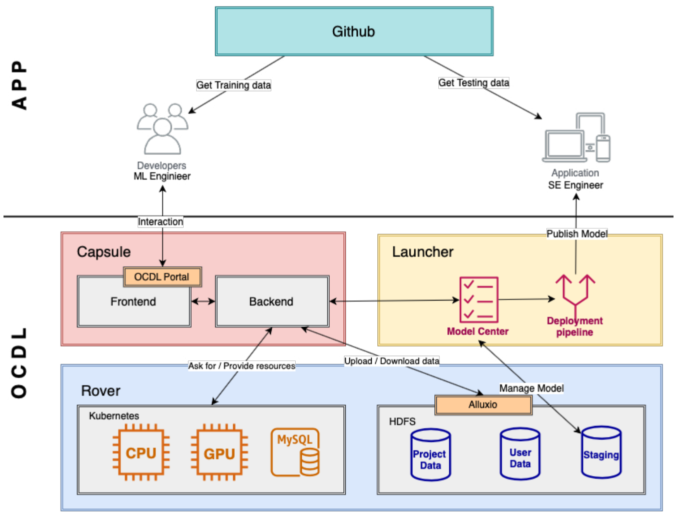


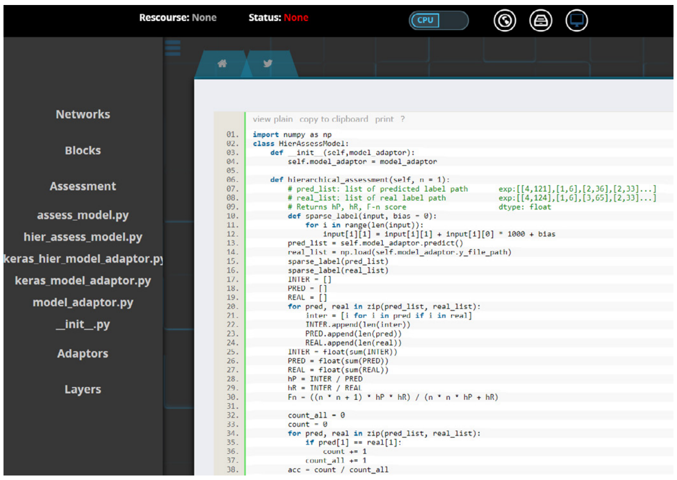
Figure 3. The OCDL architecture layers

4.1 The Capsule Layer

Capsule is a light-weight Web service that contains essential components of (1) an machine learning IDE, (2) code templates, (3) model releasing management and (4) configuration center to facilitate the development process, model operations and project resource management.

[1] https://github.com/OneClickDeepLearning

Facilitate the process of model development Capsule embeds the Jupiter IDE into its Web service as the environment for model programming, training and debugging. To facilitate programming, Capsule includes code templates that contain commonly used code snippets such as layers and adaptors. Figure 4 shows the usage of templates to quickly generate the code for revision into a proper model structure.



```python
import numpy as np
class HierAssessModel:
    def __init__(self,model_adaptor):
        self.model_adaptor = model_adaptor

    def hierarchical_assessment(self, n = 1):
        # pred_list: list of predicted label path       exp:[[4,121],[1,6],[2,36],[2,33]...]
        # real_list: list of real label path            exp:[[4,124],[1,6],[3,65],[2,33]...]
        # Returns hP, hR, F-n score                     dtype: float
        def sparse_label(input, bias = 0):
            for i in range(len(input)):
                input[i][1] = input[i][1] + input[i][0] * 1000 + bias
        pred_list = self.model_adaptor.predict()
        real_list = np.load(self.model_adaptor.y_file_path)
        sparse_label(pred_list)
        sparse_label(real_list)
        INTER = []
        PRED = []
        REAL = []
        for pred, real in zip(pred_list, real_list):
            inter = [i for i in pred if i in real]
            INTER.append(len(inter))
            PRED.append(len(pred))
            REAL.append(len(real))
        INTER = float(sum(INTER))
        PRED = float(sum(PRED))
        REAL = float(sum(REAL))
        hP = INTER / PRED
        hR = INTER / REAL
        Fn = ((n * n + 1) * hP * hR) / (n * n * hP + hR)

        count_all = 0
        count = 0
        for pred, real in zip(pred_list, real_list):
            if pred[1] == real[1]:
                count += 1
            count_all += 1
        acc = count / count_all
```

Figure 4. The example of using template codes within the Capsule IDE

Customization Capsule has a central configuration portal to specify the data source, projects, model staging information, as well as the computing resource selection. Capsule itself is deployed as a container managed by Kubernetes. Capsule and computing resources all run as Kubernetes instances. Therefore, any cloud resources can be connected to Capsule under the containerization of Kubernetes. The separation of Capsule machine learning development environment and underlying computing resources allows the customization of hybrid cloud resources and workflows for model training and releasing.

Stateless services Capsule is stateless. Any state of models and projects are communicated to the staging storage in the layer of Rover. Therefore Capsule can be deployed for each project or for a group of projects in the same way.

4.2 The Rover Layer

Rover provides computing resources and data storage.

Cross cloud resources integration Computing instances in Rover are composed of hybrid cloud resources and managed by Kubernetes. For example, a CPU and GPU clusters can be composed of Amazon EC2 instances, Azure instances or local machines. A Capsule user requests either CPU resources or GPU resources. The switch between the two types of resources can be done at any time. In addition to computing instances, Capsule is launched by Kubernetes as a containerized application. Since Capsule is stateless, the Kubernetes scheduler and self-healing capability automatically fail over any Capsule instance on CPU or GPU resources.

Data storage and retrieval Rover manages the HDFS space into three areas, namely the user space, the common space and the staging area. Each user owns a dedicated user space. A user from Capsule uploads data to the HDFS. The common space is shared among all the members of the same project. The staging area is used to store all the models submitted to the approval process. The staging area is only accessible by the project manager.

Data are fetched from the Rover HDFS to local drive of Capsule instance. Between Capsule and HDFS resides Alluxio, the data orchestration layer developed by Li (2018) as shown in Figure 3. Through the Alluxio virtual distributed storage, the access to HDFS reaches memory access speed. Data are synchornized between Alluxio and HDFS.

4.3 Launcher

Launcher builds a workflow for model evaluation, approval, releasing and deployment to end machine learning applications.

Model evaluation Model evaluation involves the similar set of entities as model development, namely trained models submitted by the model developer, test datasets and the computing resources for model inference. The inference usually takes much less delay than training. The project manager has the access to all these entities through the Capsule portal. Akin to a model developer, the manager also needs to run the submitted model on the test dataset. The code templates in Capsule provide the manager the common accessible codes such as loading models and running model inference in cross-validation of datasets.

Launcher has a component named *Model Center* that manages the states of models. When a model is submitted to the *Model Center* for evaluation, the model state is *New*. Based on the evaluation results, the manager may decide to approve or reject the model. Accordingly, the model state is set to *Approval* or *Rejected*. In the *Model Center*, the manager also categorizes the model with tags. Each model has a unique version automatically generated by the *Model Center*. All information of a model is stored in the staging area within Rover, depicted in Figure 3.

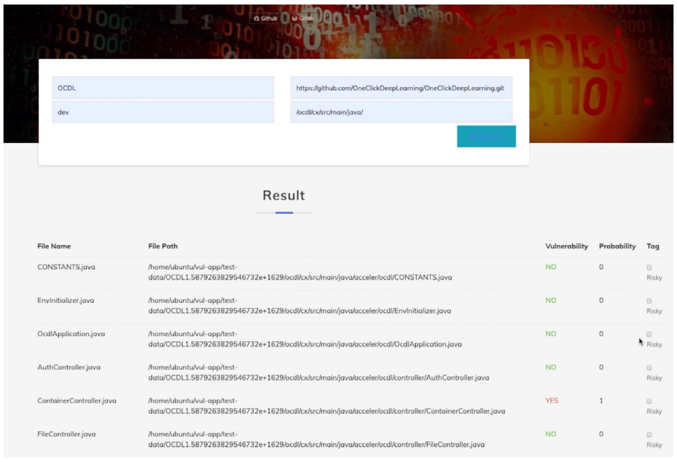
Model release Once the model is approved, it is in the process of model releasing that connects MLOps and DevOps. We adopt the publish-subscribe model for model releasing. Upon the event that the manager triggers the *release* button in the *Model Center*, the approved model is sent to AWS S3. Next, a Kafka message that wraps the model's metadata is sent to the subscribed machine learning applications. A machine learning application subscribes to the topic as the model type, rather than the model version, since new version of models of the same algorithm will be updated from time to time. When the machine learning application receives a message, it downloads the model and triggers the DevOps process as illustrated in Figure 2.

5. CASE STUDY : CODE VULNERABILITY DETECTION

OCDL has been used for developing different types of machine learning projects including NLP topic extraction and classification, medical image classification, cloud workload time series forecasting and software code vulnerability detection. In this paper, we present one case study project of detecting vulnerable software code.

Software vulnerability management is the practice of identifying, classifying, and mitigating vulnerabilities. Early detection of vulnerable software code reduces the risks of run-time errors, faults, threats and the collapse of a system. The core is to learn from vulnerable feature representations and to automate the discovery process of vulnerable cases from source code. In this case study, we have developed and trained three machine learning models over 17 open source projects in Github. Using the Capsule portal, a project account is created. Three machine learning models are coded within the Capsule Jupyter IDE. We launch CPU instances for training machine learning models of Random Forest, and Supported Vector Machine. We launch a GPU instance for training the ResNet neural network model. As shown in the top application layer of Figure 3, the training data of the 17 projects are retrieved through programming scripts from the Github APIs. The data are stored on the Rover project space.

After training models, models are submitted to the *Model Center*. Following the above process of model evaluation and model releasing in Section 4.3, models are published to the code vulnerability detection application that uses the trained model to detect vulnerable codes. The manager selects the model with the best learning performance to publish which is the random forest model with bag-of-word tokenization of software code.



OCDL
dev
https://github.com/OneClickDeepLearning/OneClickDeepLearning.git
/ocdl/cx/src/main/java/

Result

File Name	File Path	Vulnerability	Probability	Tag
CONSTANTS.java	/home/ubuntu/vul-app/test-data/OCDL1.5879263829546732e+1629/ocdl/cx/src/main/java/acceler/ocdl/CONSTANTS.java	NO	0	☐ Risky
EnvInitializer.java	/home/ubuntu/vul-app/test-data/OCDL1.5879263829546732e+1629/ocdl/cx/src/main/java/acceler/ocdl/EnvInitializer.java	NO	0	☐ Risky
OcdlApplication.java	/home/ubuntu/vul-app/test-data/OCDL1.5879263829546732e+1629/ocdl/cx/src/main/java/acceler/ocdl/OcdlApplication.java	NO	0	☐ Risky
AuthController.java	/home/ubuntu/vul-app/test-data/OCDL1.5879263829546732e+1629/ocdl/cx/src/main/java/acceler/ocdl/controller/AuthController.java	NO	0	☐ Risky
ContainerController.java	/home/ubuntu/vul-app/test-data/OCDL1.5879263829546732e+1629/ocdl/cx/src/main/java/acceler/ocdl/controller/ContainerController.java	YES	1	☐ Risky
FileController.java	/home/ubuntu/vul-app/test-data/OCDL1.5879263829546732e+1629/ocdl/cx/src/main/java/acceler/ocdl/controller/FileController.java	NO	0	☐ Risky

Figure 5. Detecting OCDL source code vulnerability

In the end-user application, we test the model using our own code of the OCDL project. As shown in Figure 5 using the OCDL platform, we are able to detect 15 classes out of 109 classes that has vulnerable code in minutes. Types of vulnerability detected are listed in Table 1.

With this case study, we further compare specific model operations among OCDL, and other machine learning

Table 1. OCDL vulnerable code detection

Item	Class Name	Vulnerabilites found
1	ContainerController	Trust boundaries Violation Unsafe Reflection Inner class could be made static
2	AuthController	Unwritten public or protected field
3	Model	Doesn't define serialVersionUID
4	ProjectService	Dead store to local variable
5	DBAlgorithmService	Dead store to local variable
6	DBProjectService	Dead store to local variable
7	DBSuffixService	Dead store to local variable
8	DBUserService	Assignment to Variable without Use Dead store to local variable
9	KafkaService	Incorrect lazy initialization of static field
10	S3Service	Incorrect lazy initialization of static field
11	AuthIntercepter	Unread field
12	ProjAuthInterceptor	Unread field
13	SecurityUtil	Unread field
14	SerializationUtils	Method ignores exceptional return value Possible null pointer dereference May fail to close stream on exception
15	Response	Inner class could be made static

Table 2. OCDL function comparision

	OCDL	Git	SageMaker	Google TFX
Model Approval	√	×	×	×
Model Evaluation	√	×	×	√
Model Releasing	√	√	√	√
Feedback	√	×	√	√

platforms including Git, AWS SageMaker and Google TFX as listed in Table 2. In addition to the model approval, evaluation, and releasing functions, we compare the operation of collecting the machine learning results and feeding the data back to improve the model. In the application of vulnerability detection, the security engineer confirms if a vulnerable code detected is true or false. Such information is stored in the Github that is later used as labelled data for fine-tuning the model.

6. CONCLUSION

The OCDL platform is built from open source frameworks that provides both the IDE and runtime for operations to develop, train, evaluate, approve, release models and finally deploy models to the hosting machine learning applications. Our design of the OCDL platform is driven by roles involved and the lifecycle of machine learning development. The OCDL supports virtual resource provision and HDFS data storage access at the memory level speed. We demonstrate this platform with an end-to-end deployment of machine learning models to detect vulnerable code of our own OCDL development. Our future work includes automated deployment of trained models as services to machine learning applications.

REFERENCES

Chang, X. and Zha, L. (2018). The performance analysis of cache architecture based on alluxio over virtualized infrastructure. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 515–519.

Ciucu, R., Adochiei, F., Adochiei, I., Florin, A., Seritan, G., Enache, B.A., Grigorescu, S., and Argatu, V. (2019). Innovative devops for artificial intelligence. *The Scientific Bulletin of Electrical Engineering Faculty*, 19, 58–63.

de Oliveira Werneck, R., de Almeida, W.R., Stein, B.V., Pazinato, D.V., Júnior, P.R.M., Penatti, O.A.B., Rocha, A., and da Silva Torres, R. (2018). Kuaa: A unified framework for design, deployment, execution, and recommendation of machine learning experiments. *Future Generation Computer Systems*, 78, 59 – 76.

Erich, F.M.A., Amrit, C., and Daneva, M. (2017). A qualitative study of devops usage in practice. *Journal of Software Evolution Process*, 29(6).

Krishnamurthi, R., Maheshwari, R., and Gulati, R. (2019). Deploying deep learning models via iot deployment tools. In *2019 Twelfth International Conference on Contemporary Computing (IC3)*, 1–6.

Li, H. (2018). *Alluxio: A Virtual Distributed File System*. Ph.D. thesis, EECS Department, University of California, Berkeley.

López García, , De Lucas, J.M., Antonacci, M., Zu Castell, W., David, M., Hardt, M., Lloret Iglesias, L., Moltó, G., Plociennik, M., Tran, V., Alic, A.S., Caballer, M., Plasencia, I.C., Costantini, A., Dlugolinsky, S., Duma, D.C., Donvito, G., Gomes, J., Heredia Cacha, I., Ito, K., Kozlov, V.Y., Nguyen, G., Orviz Fernández, P., Šustr, Z., and Wolniewicz, P. (2020). A cloud-based framework for machine learning workloads and applications. *IEEE Access*, 8, 18681–18692.

Medel, V., Rana, O., Bañares, J.a., and Arronategui, U. (2016). Modelling performance resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, 257–262. Association for Computing Machinery, New York, NY, USA.

Miao, H., Li, A., Davis, L.S., and Deshpande, A. (2017). Modelhub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 1393–1394.

Rothenhaus, K., De, Soto, K., Nguyen, E., and Millard, J. (2018). Applying a developmentoperations (devops) reference architecture to accelerate delivery of emerging technologies in data analytics, deep learning and artificial intelligence to the afloat u.s. navy.

Surya, R.Y. and Imam Kistijantoro, A. (2019). Dynamic resource allocation for distributed tensorflow training in kubernetes cluster. In *2019 International Conference on Data and Software Engineering (ICoDSE)*, 1–6.

Wang, M., Zhang, D., and Wu, B. (2020). A cluster autoscaler based on multiple node types in kubernetes. In *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, volume 1, 575–579.

Zou, Y., Jin, X., Li, Y., Guo, Z., Wang, E., and Xiao, B. (2014). Mariana: Tencent deep learning platform and its applications. *Proc. VLDB Endow.*, 7(13), 1772–1777.