Model-Driven Approach to Blockchain-Enabled MLOps

Nenad Petrović

***Abstract*—In recent years, machine learning has reached quite sophisticated level of usability within applications across various domains – ranging from booking reservations and media content delivery to business and healthcare. However, the deployment of machine learning models, together with parameter tuning and periodic training, which are necessary to maintain satisfiable performance, represent time consuming processes, requiring various types of skills - both DevOps and data analysis-related. In this paper, we leverage model-driven approach in synergy with code generation with aim to automatize the so-called MLOps activities, relying on ZenML framework for pipeline automation and Kubernetes for containerized task orchestration. On top of that, we leverage Blockchain for infrastructure provisioning. Our goal is to reduce the cognitive load of infrastructure and services management within systems relying on machine learning. The framework is evaluated in scenarios using PyTorch-based deep learning predictive models. According to the results, the proposed approach reduces both the time and skill required for successful MLOps activities.**

***Index Terms*—DevOps; Kubernetes; MLOps; PyTorch; Blockchain.**

I. Introduction

Continuous integration and delivery have become standard in software engineering workflow within the last decade. The goal of so-called DevOps practice is to align the deployment of software artifacts with business goals which are enabled by them, so the customer's organization can benefit from them as quickly as possible. However, operations related to underlying infrastructure management are becoming more and more complex, due to heterogeneity of services, devices and increasing performance demands. Therefore, due to fact that machine learning (ML) services are recognized as crucial enablers of novel usage scenarios across various domains, a distinct subfield with focus on them has emerged, known as MLOps [1-4]. It is an extension of now well-established DevOps paradigm with aspects specific to service delivery in machine learning, such as continuous model training for prediction performance improvement, rapid deployment and parameter tunning towards automated generation of complex ML task pipelines [1-4].

In this paper, the focus is on reducing the complexity of MLOps-related activities and service delivery relying on model-driven approach [5]. Moreover, the business-related aspects of infrastructure resource provisioning and usage charging using blockchain by the provider are also considered. The main contributions of this paper are the following: 1) MLOps metamodel – defining the structure of user-created model instances representing machine learning pipelines with several distinct steps together with aspects related to its deployment 2) code generator – leverages the model for automated code generator covering several aspects: pipeline script, predictive model, infrastructure management 3) blockchain-based transaction model making use of smart contracts for renting high-performance computing resources aiming accelerated machine learning.

In our previous works, metamodel-based approach in synergy with ontologies was leveraged for automated container-based service deployment in Fog Computing [6]. On the other side, a similar method was adopted in [7] for generation of predictive models starting from high-level predictive problem descriptions aiming state-of-art mobile network infrastructure planning and management.

Nenad Petrović is with the Faculty of Electronic Engineering, University of Niš, Aleksandra Medvedeva 14, 18000 Niš, Serbia (e-mail: nenad.petrovic@elfak.ni.ac.rs), (https://orcid.org/0000-0003-2264-7369)

II. Background

A. ZenML

ZenML [8] is open-source, high-level Python framework for machine learning pipeline automation. It is available as Python library in form of function decorators and specific classes inside scripts, while it imposes pre-defined code structure. The overview of main ZenML-related concepts and terminology is given in Table I.

TABLE I
ZenML Concepts Overview

Concept	Description
Repository	A special type of directory, declared used `zenml init` command. Each ZenML action must take place within a repository, which is created
Step	Single stage within ML flow, representing a node of in ML flow computation graph. Implementation-wise, they represent Python functions with typed parameters in signature for both arguments and return value. Decorator used is `@step`, while step result caching can be enabled using `enable_cache=True` parameter
Pipeline	A sequence of steps. It connects all the steps, their inputs and outputs. Decorator used is `@pipeline`. Moreover, it is possible to set the list of external libs/dependencies from .txt file can be done using `requirements_file` attribute of the decorator. It is run by invoking `pipeline.run()` method inside Python script. Optionally, a scheduler object can be assigned for periodic execution of certain steps enabling scenarios such as continuous model training.

 ISBN 978-86-7466-930-3

Stack	Represents environment and configuration of MLOps platform infrastructure. It consists of: artifact store, metadata store, container registry, orchestrator and custom step operators. Creating a new stack with desired parameters is done using `stack register`, while it is run using `stack up` command., which has to be done before running any pipeline.
Artifact store	Persistent storage of step results.
Materializer	Defines how data is passed between steps. Serialization and deserialization are used while storing/retrieving results.
Metadata store	Keeps the data related to pipeline, step and experiment configuration and references for tracking of inputs/outputs within artifact store created within ML pipeline.
Orchestrator	Component for scheduling and running pipeline steps
Container registry	Stores Docker images required for running the steps
Custom step operator	User-defined environments for running ML flow tasks within Docker containers
Integration	Enable usage of various third-party tools enriching ML development like pytorch, tensorflow and sklearn. Apart from that, it also includes orchestrator-enabled stacks, such as local-kubeflow (Kubernetes-based) and airflow.

Depiction of ZenML architecture showing how the previously mentioned concepts are related is given in Fig. 1.

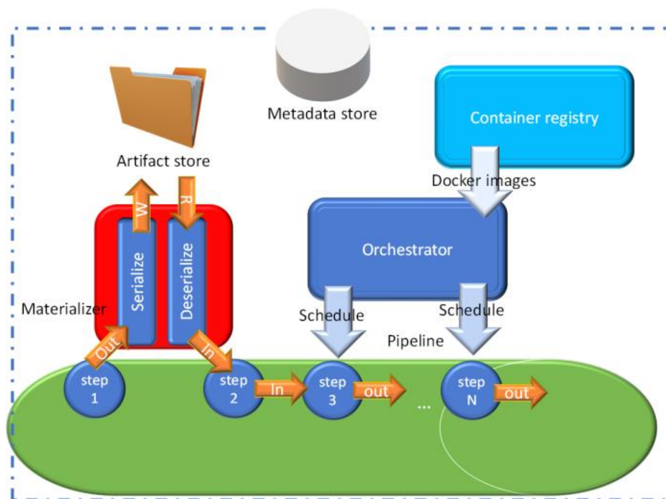

Fig. 1. ZenML concepts and their relations.

Additionally, Fig. 2 shows the programming workflow using ZenML. First, we initialize a repository inside the desired directory where we place our Python script. After that, in Python code we define the typical steps of ML flow: 1) importer – downloading and loading dataset 2) trainer – passing through dataset and updating model weights for new predictions 3) evaluator – estimates how good the prediction performance is, according to the given metric (accuracy for classification; mean relative error - MRE for regression). Moreover, we connect the steps in a sequence as shown within the pipeline object and finally run the pipeline object instance.

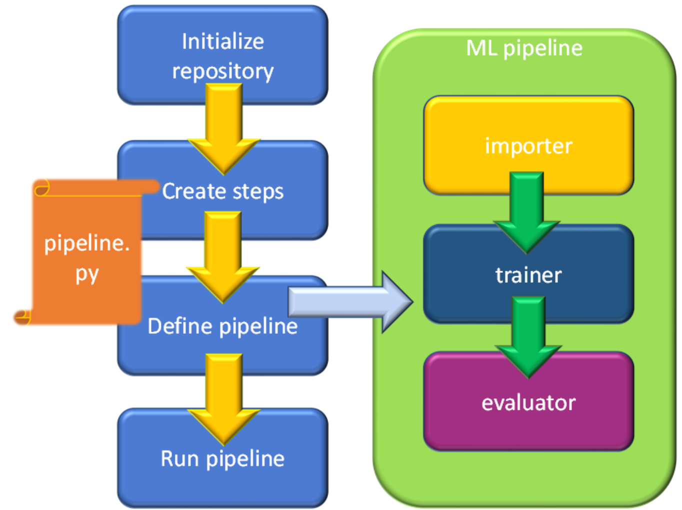

Fig. 2. ML pipeline creation using ZenML in Python.

B. *Kubernetes*

Kubernetes [9] represents an open-source platform whose goal is to enable deployment and management of containerized services run on multi-server clusters. Furthermore, it provides convenient access to useful features, such as scalability, fault-tolerance and declarative configuration. Table III gives an overview of key concepts within Kubernetes-based architectures.

TABLE III
KUBERNETES CONCEPTS OVERVIEW

Concept	Description
Control plane	Cluster management component, responsible for global decisions, and scheduling
Node	Worker machines within the cluster running containerized apps
Pod	Smallest deployable unit, which consists of one or more app containers. These containers share storage, network specification and config. Optionally, might include data volumes for persistent storage
Service	A logical set of pods
Kubectl	Comamnd-Line Interface (CLI) tool for running commands against Kubernetes cluster, such as: -Deployment (`kubectl apply deployment.json` and `kubectl create deployment dep_name --image=docker_image` -Scaling up/down (`kubectl scale --replicas=num resource_name`) -Retrieval of node, pod and service info (`kubectl get pods, nodes, services`)

Despite the fact that Kubernetes provides automatic scheduling capabilities, there might be situations where deployment of pod has to be done on a specific node. In that case, we leverage node labels. The corresponding command for labelling a node has the following form: `kubectl label nodes <node_name> label_name=label_value`. After that, when we want to create a pod using YAML configuration file, it would be necessary to make use of `nodeSelector` property and set it as `label_name:label_value`.

 ISBN 978-86-7466-930-3

Kubernetes-based architecture is depicted in Fig. 3. In this paper, we make use of containerized custom step operators in ZenML run as Kubernetes pods, which are actually containers running PyTorch code. Moreover, node labels are leveraged in order to have low-level scheduling control and determine where each of these steps will be executed, enabling additional flexibility.

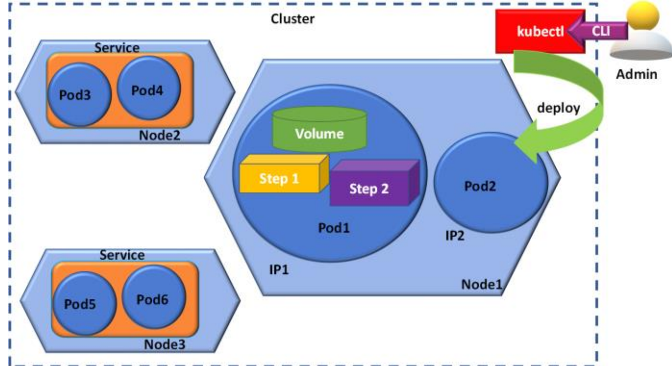

Fig. 3. Kubernetes-based containerized app architecture.

III. Implementation

A. Workflow Overview

Model-driven workflow of the proposed framework for automated ML-related task deployment is illustrated in Fig. 4. In the first step, user creates a deployment diagram model instance which describes the ML pipeline and underlying execution infrastructure details. After that, the model is parsed, so code generator constructs Python script containing the ML pipeline code relying on ZenML and PyTorch in synergy with numpy. Moreover, corresponding Kubernetes orchestrator commands are generated in order to ensure that distinct pipeline steps are executed on the desired cluster nodes. Additionally, the allocated resources are charged to the customer by parametrizing smart contracts for blockchain-based transactions, while the price might vary due to presence of deep learning accelerator cards on some of the nodes. Finally, the generated machine learning Python script is executed on the allocated computing nodes.

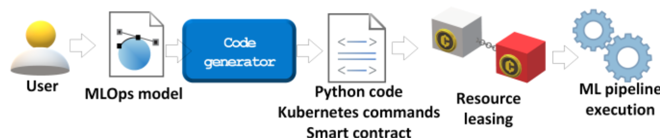

Fig. 4. Blockchain-enabled model-driven MLOps workflow.

B. MLOps Metamodel

When it comes to adoption of model-driven engineering, we make use of metamodel which defines the structure of user-created deployment diagram (shown in Fig. 5). For implementation, Ecore [10] within Eclipse Modelling Framework (EMF) [11] in Java is used, which automatically generates all the auxiliary classes for model manipulation, together with convenient GUI-enabled editor.

The highest-level concept is *Pipeline*, which consists of one or more machine learning tasks, referred to as *Step*. A Pipeline can be executed periodically for purpose of continuous training, which is defined by *repeatTime* property. Moreover, each of the Steps can be one of the following type with specific, distinct properties: *Importer*, *Trainer* or *Evaluator*. Importer represents ML task which downloads the corresponding dataset and opens the downloaded file. In this context, it is necessary to set URL corresponding to the location where dataset is stored online, denoted as *onlineData*. Otherwise, if dataset is local and already present on disk, another parameter is used – *localData*. When it comes to trainer step, it is possible to set its learning rate, number of batches, select the target implementation technology, but we make use of PyTorch in this paper. Each trainer can use pre-created model, given by *modelPath* or it is necessary to define a custom neural network. For custom neural network, its architecture is described using *Layer* element, while each of them has type (such as Convolutional – in image classification or standard Fully Connected in Multi-Layer Perceptron), number of processing units (neurons) and activation function (such as ReLU, softmax, sigmoid). Finally, the performance metric used within Evaluator step depends on the type of machine learning task, and we cover two possibilities relevant to supervised learning as *predictionType* property of *Pipeline* – classification (*Accuracy*) and regression (*Mean Relative Error*).

On the other side, the aspects of distinct Step deployment are covered by the metamodel as well. For each pipeline part, there is an attribute *targetLabel*, describing which worker node within Kubernetes cluster would be preferred for execution of pod created within custom step operator. Additionally, infrastructure executing the pipeline is represented as *Cluster* that consists of *Nodes*. For each *Node*, the following properties are customizable, such as label, location, IP address, accelerator (whether it has dedicated hardware for deep learning attached) and unit price (depending on the node performance).

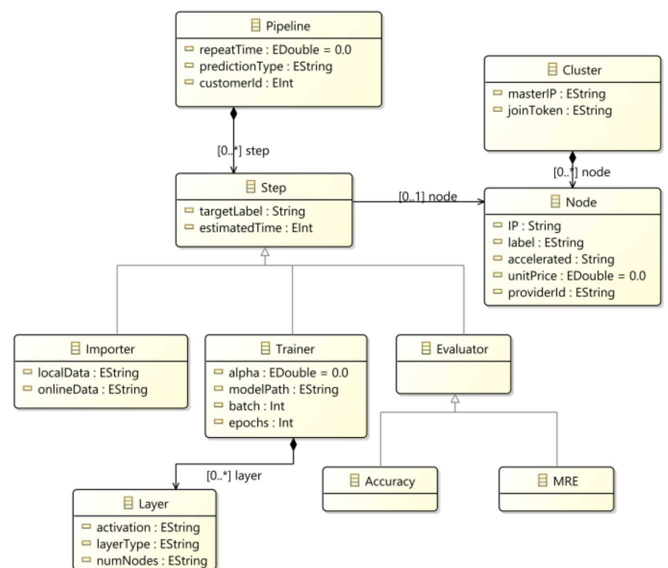

Fig. 5. UML class diagram of MLOps metamodel.

 ISBN 978-86-7466-930-3

C. Code Generation

The user-drawn deployment model is first parsed and then traversed relying on Ecore-generated classes. Pipeline element and iterated for each of the contained steps. When it comes to each step, it is necessary to get the target label and insert it into nodeSelector section within Kubernetes deployment YAML file. After that, depending on the step type, corresponding template is used. For importer steps, it is only necessary to change the path where pre-created model is located. When it comes to trainer, it contains model training loop iterating for given number of epochs through batch number of dataset samples and desired learning rate (α) value. For evaluator step, the corresponding method is chosen according to the type of prediction problem. Additionally, the leasing of resources is charged to the customer by valorizing blockchain smart contract with unit price of the selected node. Finally, the previously created Kubernetes YAML describing step deployment is applied, so the pod with Docker container running ML task inside is spawned on the selected node. Pseudocode of the code generation procedure is given in Table IV.

TABLE IV
CODE GENERATION PSEUDOCODE

Input: MLOps deployment model
Output: Python script, Kubernetes commands, Smart contract
Steps:
1. deployment.elements:=parse(model);
2. Retrieve pipeline from deployment.elements;
3. For each step in pipeline
4. Create nodeSelector for step.targetLabel;
5. If(step is Importer)
6. Generate importer loading dataset from step.localData or onlineData;
7. If(step is Trainer)
8. Load model from step.modelPath;
9. Generate TrainerCode(step.epochs, step.batch, step.alpha);
10. If(step is Evaluator)
11. If pipeline.predictionType is regression
12. Use Mean Relative Error;
13. Else
14. Use Accuracy;
15. Get node.unitPrice;
16. Calculate total leasing price as step.estimatedTime*step.node.unitPrice
17. Genrate smart contract between pipeline.customerId and step.node.providerId for total price;
18. Apply Kubernetes deployment for pod with step.id;
19. End for each
20. End

D. ZenML pipeline Relying on PyTorch Deep Learning Models

Deep learning refers to approach in artificial intelligence making use of neural networks with one or many hidden layers between the inputs and outputs. PyTorch [12] is library for Python which covers the required set of capabilities for deep learning: tensor manipulation and high-level object-oriented representation of both models and datasets. In Fig. 6, an excerpt of typical ZenML pipeline relying on PyTorch models is given. This kind of Python script actually represents one of the outputs of code generator. When it comes to neural network models in PyTorch, their capabilities are encapsulated within *Module* class which has to be inherited by any custom model. In this class, within the constructor we define neural network architecture (layers, nodes and activation functions), while *forward* connects the layers defining how data passes through neural network. In given example, we use MNIST dataset [13] of handwritten digits for purpose of classification.

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.flat_network = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 311),
            nn.ReLU(),
            nn.Linear(311,10)
        )
        # fully connected layer, output 10 classes
        self.out = nn.Linear(10, 10)

    def forward(self, x):
        x = torch.unsqueeze(x, dim=0)
        x = self.flat_network(x)
        x = self.out(x)
        output = self.out(x)
        return output

def get_data_loader_from_np(X: np.ndarray, y: np.ndarray) -> DataLoader:
    tensor_x = torch.Tensor(X)  # transform to torch tensor
    tensor_y = torch.Tensor(y).type(torch.LongTensor)

    torch_dataset = TensorDataset(tensor_x, tensor_y)
    torch_dataloader = DataLoader(torch_dataset)
    return torch_dataloader

@step(custom_step_operator="trainer1", enable_cache=False)
def torch_trainer(
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> nn.Module:

    train_loader = get_data_loader_from_np(x_train, y_train)

    model = Net().to(DEVICE)
    optimizer = optim.Adadelta(model.parameters(), lr=0.001)

    scheduler = StepLR(optimizer, step_size=1, gamma=0.01)
    for epoch in range(1, num_epochs):
        model.train()
        for batch_idx, (data, target) in enumerate(train_loader):
            data, target = data.to(DEVICE), target.to(DEVICE)
            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            loss.backward()
            optimizer.step()
        scheduler.step()

    return model

@step(custom_step_operator="evaluator1", enable_cache=False)
def classification_evaluator(
    X_test: np.ndarray,
    y_test: np.ndarray,
    model: nn.Module,
) -> float:
    model.eval()
    test_loader = get_data_loader_from_np(x_test, y_test)
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(DEVICE), target.to(DEVICE)
            output = model(data)
            test_loss += F.nll_loss(
                output, target, reduction="sum"
            ).item()
            pred = output.argmax(
                dim=1, keepdim=True
            )
            correct += pred.eq(target.view_as(pred)).sum().item()

    return correct / len(test_loader.dataset)

@step(custom_step_operator="importer1", enable_cache=False)
def my_importer() -> Output(
    x_train=np.ndarray, y_train=np.ndarray, x_test=np.ndarray, y_test=np.ndarray
):
    (X_train, y_train), (
        X_test,
        y_test,
    ) = load_data(dataset_path)
    return x_train, y_train, x_test, y_test

@pipeline(required_integrations=[PYTORCH])
def my_pipeline(
    importer,
    trainer,
    evaluator,
):
    x_train, y_train, x_test, y_test = importer()
    model = trainer(x_train=x_train, y_train=y_train)
    evaluator(x_test=x_test, y_test=y_test, model=model)

continous_train = Schedule(
     start_time = datetime.now(),
     end_time = datetime.now() + timedelta(minutes = 5),
     interval_second = 60
)

if __name__ == "__main__":
    torch_pipeline = my_pipeline(
        importer=my_importer(),
        trainer=torch_trainer(),
        evaluator=classification_evaluator(),
    )

    torch_pipeline.run(schedule = continous_train)
```

Fig. 6. ZenML PyTorch training script for MNIST dataset.

 ISBN 978-86-7466-930-3

E. Resource Leasing Relying on Solidity Smart Contract

Blockchain enables decentralized approach to immutable and irreversible transactions relying on approval by huge network of computer nodes, making it secure and reliable. On the other side, smart contracts define actions executed within protocol for realization of blockchain-based transaction. In this paper, we make se of Ethereum blockchain in synergy with Solidity smart contracts [14]. Solidity code of the underlying transaction mechanism for resource leasing in context of ML task execution is given in Fig. 7. As it can be seen, the information stored as part of transaction consists of *customerId*, *providerId* and identifier of node which will execute some ML task which represents a step within pipeline. First, the total price is calculated by multiplying *unitPrice* and *estimatedTime* required for step execution. After that, the transaction itself is performed by transferring the previously calculated total amount of tmokens from customer's to provider's account.

```
contract LeasingInfrastructure {
   address public providerId;
   uint32 public nodeId;
   uint32 public stepId;
   mapping (address => uint) public balances;

   event Sent(address customerId, address providerId, uint total);

   function leaseNode(address received, uint unitPrice, uint estimateTime) public {
      total = uintPrice*estimatedTime;
       require(total <= balances[msg.sender], "Not enough tokens");
      balances[msg.customerId] -= total;
      balances[providerId] += total;
      emit Sent(msg.customerId, providerId, total);
   }
}
```

Fig. 7. Solidity smart contract for ML task resource leasing.

IV. EXPERIMENTS AND EVALUATION

For evaluation of the proposed framework, three publicly available image classification datasets were used. The first two tackle image classification problem: 1) yoga pose determination (our previous work presented in [15]) - 5 poses in dataset of 1551 images 2) MNIST [13] – 70 000 images of handwritten digits 0-9. On the other side, a regression problem of service demand prediction in telco networks from [7] was considered as the third case. In all of the experiments, test was 20% of the overall dataset with no overlapping samples from training set. The presented experiments were run on MacBook Pro (16-inch, 2019) laptop, equipped with 2.3GHz 8-core Intel Core i9 CPU, 16GB of DDR4 memory, 1TB SSD and Intel UHD Graphics 630 with 1.5GB VRAM. On the other side, Kubernetes cluster consisted of two more Ubuntu machines equipped with Intel i5 CPU, 8GB DDR4 RAM and 4GB GPU.

The results of the experiments are given in Table IV. Several aspects were considered: code generation time, model training time, speed-up compared to manual pipeline creation including model creation (moderately experienced machine learning engineer) and achieved prediction performance (MRE for regression, accuracy for classification).

TABLE IV
EXPERIMENT RESULTS

Case	Code generation [s]	Model training [s]	Speed-up [times]	Performance [%]	Manual pipe [s]
Yoga pose [15]	0.911	317	45	Accuracy 73%	104
MNIST [13]	0.87	124	36	Accuracy 96%	91
Telco [7]	0.93	27	21	MRE 9%	88

As it can be seen, in all the cases, the achieved speed-up was more than 20 times compared to traditional approach involving manual Python code writing from scratch. However, the speed-up is more significant is case of more complex models based on convolutional neural networks with huge number of layers – it was yoga pose determination. In our case, the only manual operation is pipeline deployment model creation using GUI tool, which took about 1.5 minutes in our experiments. All the models show almost identical performance to traditional counterparts, as expected. When it comes to code generation, execution time does not exceed 1 second in the presented case studies. Finally, the overhead of model training compared to execution without MLOps framework is around 15% when run on single machine and k3d [16] local Kubernetes cluster, but can be compensated by smart scheduling techniques, especially for larger datasets.

V. CONCLUSION AND FUTURE WORK

According to the achieved experimental results, the proposed model-driven approach to MLOps leveraging automated code generation further speeds up the development of machine learning services, required administration operations and their delivery to the customers. Moreover, it also accelerates resource leasing protocols adopting blockchain-based smart contracts for transactions and their automated generation. Finally, the adoption of intuitive model-driven tools opens new horizons of machine learning service adoption and management even by persons without expertise in this area.

However, there are several possible research directions in future. First, we would work on integration of model-driven resource allocation mechanisms relying on multi-objective optimization approach [17] for energy and cost-efficient ML pipeline task scheduling. Moreover, the incorporation more sophisticated federated learning mechanisms and neural network layer splitting strategies across multiple cluster nodes aiming time-critical scenarios would be considered as well.

ACKNOWLEDGMENT

This work has been supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia.

REFERENCES

[1] G. Symeonidis, E. Nerantzis, A. Kazakis and G. A. Papakostas, "MLOps - Definitions, Tools and Challenges," 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), pp. 453-460, 2022. https://doi.org/10.1109/CCWC54503.2022.9720902

[2] S. Moreschini, F. Lomio, D. Hästbacka, D. Taibi, "MLOps for evolvable AI intensive software systems", IEEE International Conference on Software Analysis, Evolution and Reengineering 2022, pp. 1-2, 2022.

[3] D. Kreuzberger, N. Kühl, S. Hirschl, "Machine Learning Operations (MLOps): Overview, Definition, and Architecture", preprint, 2022.

[4] D. A. Tamburri, "Sustainable MLOps: Trends and Challenges", 2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 17-23, 2020. https://doi.org/10.1109/SYNASC51798.2020.00015

[5] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, 2nd Edition, Morgan & Claypool Publishers, 2017.

[6] N. Petrovic, M. Tosic, "SMADA-Fog: Semantic model driven approach to deployment and adaptivity in Fog Computing", Simulation Modelling Practice and Theory, 102033, pp. 1-25, 2019. https://doi.org/10.1016/j.simpat.2019.102033

[7] D. Krstić, N. Petrović, I. Al-Azzoni, "Model-Driven Approach to Fading-Aware Wireless Network Planning Leveraging Multiobjective Optimization and Deep Learning", Mathematical Problems in Engineering, vol. 2022, 4140522, Special Issue: Mathematical Modelling of Data Transmission in Next Generation Wireless Systems, 2022, pp. 1-23, 2022. https://doi.org/10.1155/2022/4140522

[8] ZenML [online]. Available on: https://zenml.io/, last accessed: 08/05/2022.

[9] Kuberentes [online]. Available on: https://kubernetes.io/, last accessed: 08/05/2022.

[10] Eclipse Modeling Framework [online]. Available on: https://www.eclipse.org/modeling/emf/, last accessed: 08/05/2022.

[11] Ecore [online]. Available on: https://wiki.eclipse.org/Ecore, last accessed: 08/05/2022.

[12] E. Stevens, L. Antiga, T. Viehmann, *Deep Learning with PyTorch*, Manning Publications, 2020

[13] The MNIST database of handwritten digits [online]. Available on: http://yann.lecun.com/exdb/mnist/ , last accessed: 08/05/2022.

[14] Solidity [online]. Available on: https://docs.soliditylang.org/en/v0.8.13/, last accessed: 08/05/2022.

[15] M. Radenković, V. Nejković, N. Petrović, "Adopting AR and Deep Learning for Gamified Fitness Mobile Apps: Yoga Trainer Case Study", AIIT 2021 International conference on Applied Internet and Information Technologies, pp. 167-171, 2021.

[16] K3d [online]. Available on: https://k3d.io/v5.4.1/ , last accessed: 08/05/2022.

[17] I. Al-Azzoni, J. Blank, N. Petrović, "A Model-Driven Approach for Solving the Software Component Allocation Problem", Algorithms 2021; 14(12):354, pp. 1-19, 2021. https://doi.org/10.3390/a14120354