AIoTwin
Twinning action for spreading excellence in Artificial Intelligence of Things

DELIVERABLE D1.3

AIOTWIN MIDDLEWARE IMPLEMENTATION AND EVALUATION

Funded by the European Union

Project number: 101079214
Project name: Twinning action for spreading excellence in Artificial Intelligence of Things
Project acronym: AIoTwin
Call: HORIZON-WIDERA-2021-ACCESS-03
Topic: HORIZON-WIDERA-2021-ACCESS-03-01
Type of action: HORIZON Coordination and Support Actions
Granting authority: European Research Executive Agency

DOCUMENT CONTROL

Deliverable No.	**D1.3**
Title	AIoTwin Middleware Implementation and Evaluation
Lead Editor	Ivan Čilić
Type	Report
Dissemination Level	PU/SEN
Work Package	WP1
Due Date	31.05.2024.

AUTHOR(S)

Name	Partner	e-mail
Ivana Podnar Žarko	UNIZG-FER	ivana.podnar@fer.hr
Ivan Čilić	UNIZG-FER	ivan.cilic@fer.hr
Anna Lackinger	TUW	a.lackinger@dsg.tuwien.ac.at
Anh Le-Tuan	TUB	anh.letuan@tu-berlin.de

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
0.01	22-04-2024	Ivan Čilić	Initial structure
0.02	07-05-2023	Ivan Čilić	Learning pipeline: architecture and mechanisms
0.03	10-05-2023	Ivan Čilić	Learning pipeline: implementation
0.04	13-05-2024	Anna Lackinger	Learning pipeline: FL service implementation and pilot experiments
0.05	14-05-2024	Ivan Čilić	Inference pipeline, introduction and conclusion
0.06	17-05-2024	Ivana Podnar Žarko	Internal review of the entire document
0.07	20-05-2023	Anna Lackinger, Ivan Čilić	Applying changes based on first internal review
0.08	24-05-2024	Anh Le-Tuan	Internal review of the entire document
0.09	29-05-2024	Ivan Čilić	Applying changes based on second internal review
1.00	31-05-2024		Final version submitted to EC

Table of Contents

Executive Summary

This deliverable outlines the current status of the implementation and evaluation of the AIoTwin middleware, a data-driven orchestration middleware for energy-efficient IoT supporting machine learning (ML) workflows. The implementation includes the following: (i) the learning pipeline and (ii) the inference pipeline. The learning pipeline, on the one hand, facilitates an adaptive orchestration of federated learning workflows. This involves deploying components of the federated learning pipeline, monitoring its execution, and performing reconfiguration as necessary. To achieve this, an orchestration mechanism has been developed as an extension of the Kubernetes container orchestration tool. The effectiveness of this implementation has been assessed across both simulated and real-world infrastructure settings. The inference pipeline, on the other hand, relies on QoS-aware data routing within the Edge-to-Cloud Continuum (ECC). To support this, a distributed QoS-aware load balancer named QEdgeProxy has been introduced. QEdgeProxy dynamically manages a pool of service instances within the ECC that meet the Quality of Service (QoS) requirements for a given service, and it forwards service requests to these instances while also balancing the load across them. Implemented within the Kubernetes ecosystem, QEdgeProxy has been evaluated within a realistic K3s cluster environment in a dynamic scenario which includes instances failures and network changes. The experimental results demonstrate that QEdgeProxy outperforms both Kubernetes' built-in mechanisms and a comparable state-of-the-art solution, all while introducing minimal computational overhead.

1 Introduction

1.1 Deliverable Context and Description

Work Package 1 (WP1) comprises the tasks and activities of the AIoTwin Joint Research Project, which incorporates the contributions and expertise of all AIoTwin partners. In this joint research, we are focussing on the following goals:

- Develop a data-driven orchestration middleware for energy-efficient IoT supporting ML workflows.
- Investigate the mechanisms for orchestration of containers across the edge-to-cloud continuum.
- Run and test the middleware on the available infrastructure of all consortium partners.

In the preceding deliverable, D1.1, we presented the use cases, system requirements, and general architecture of the AIoTwin data-driven orchestration middleware. Subsequently, in deliverable D1.2, we introduced a demonstration of relevant state-of-the-art mechanisms for edge orchestration, energy efficiency, and machine learning workflows. Building upon these deliverables, this deliverable serves as a direct continuation, focusing on the practical implementation of the outlined requirements and proposed architecture. In addition, the decisions regarding the selection of the applied federated learning frameworks and orchestration tools and mechanisms were based on the conclusions provided by the demonstrator. The resulting middleware implementation has been made publicly available as open-source software via the AIoTwin GitHub repository (https://github.com/aiotwin).

This is the first version of the deliverable on AIoTwin middleware implementation and evaluation. The second and final version is planned for delivery in M33 (D1.7).

1.2 Deliverable Outline

This document is organized as follows. Section 2 introduces the AIoTwin middleware and identifies the requirements that were implemented so far. Section 3 provides the implementation and evaluation of the adaptive orchestration of FL workflows that supports the learning pipeline of the middleware. Section 4 provides the implementation and evaluation of the QoS-aware data routing in Edge-to-Cloud Continuum (ECC) that targets the inference pipeline of the middleware. Section 5 concludes the deliverable and Section 6 contains a list of acronyms used. A list of figures, tables and references can be found at the end of the document.

2 AIoTwin Middleware

In D1.1, we presented the use cases, system requirements, and general architecture of the AIoTwin data-driven orchestration middleware. The architecture in D1.1 defines the components for managing machine learning (ML) tasks, including both learning and inferring phases. Consequently, D1.3 presents the implementations of the components and reports their current implementing status. The ongoing implementation can be divided into two parts:

- **Adaptive orchestration of federated learning workflows**
  - This part implements the learning pipelines and addresses the following requirements which were introduced in D1.1:

No.	Description
1	Middleware should efficiently manage and monitor resources on each node.
2	Middleware should collect the distribution of data available on node for ML.
3	Middleware should collect the information on the underlying network connecting nodes in the ECC.
4	Middleware should deploy and manage services across the ECC.
5	Middleware should be able to run a configuration model to output configuration of a ML pipeline.
6	Middleware should deploy ML components based on a learning configuration.
7	Middleware should monitor learning performance.
8	Middleware should reconfigure the learning pipeline if a better learning performance can be achieved.

- **QoS-aware data routing in ECC**
  - This part implements the inference pipeline and addresses the middleware requirement:

No.	Description
13	Middleware should maintain a desired QoS for clients using the inference services.

The remaining requirements related to energy efficiency and management of inference services (9-12) will be addressed in the next version which will introduce more advanced features.

3 Learning Pipeline: Adaptive Orchestration of Federated Learning Workflows

With the rapid growth of IoT devices, enormous amounts of data are constantly generated. The data can be used to train ML models to improve IoT services. The traditional ML pipelines assume that data is collected and trained in a centralized cloud location. However, these approaches come with privacy concerns, as the data is exposed to the training services and also cloud providers. Federated learning (FL) is proposed to deal with these privacy concerns by allowing the model to be trained without the need to centrally store the data [1][2]. The collected data remains on the owners' storage (in FL they are referred to as *clients*) as only the trained weights are sent to the central coordinating server for updating the model (model aggregation). This approach significantly reduces the privacy issues and reduces the network bandwidth as the raw data is not sent out of the originating devices. In addition, by distributing the training task across different devices, the computational and storage costs of the central server will be significantly reduced.

However, as highlighted in [3] and in deliverable D1.1, FL has several open challenges. One of the major challenges of FL is the heterogeneity of its participants (clients) in terms of hardware, network and data distribution. Hardware heterogeneity leads to performance degradation and occurrences of *stragglers* [4]. The distributed architectures in FL also introduce the problem of network heterogeneity, particularly in IoT settings with unstable networks. Obviously, when the model is big (several gigabytes), performance of FL training highly depends on communication efficiency [5]. Hierarchical FL is proposed to reduce communication costs by placing multiple local aggregators at the edge of the network, closer to the FL clients, to perform local aggregation before sending the aggregated models to the global aggregator [5]. Data distribution poses another challenge, as performance degradation is observed in non-IID data scenarios [6]. Therefore, in hierarchical FL, balancing clusters by considering node data distribution is crucial, as discussed in [7].

The overview of FL challenges brought us to the following observations:

- Defining the configuration of an FL pipeline is highly dependent on the model to be trained, the underlying infrastructure, and the data distribution.
- Due to the dynamicity of the edge environment during the execution of FL, the configuration of an FL pipeline is likely to be changed during runtime.

Regarding the above observations, an adaptive orchestration mechanism for FL pipeline in an ECC environment is required. The adaptive mechanism allows the orchestrator to monitor the execution of the FL pipeline, and perform reconfiguration as needed. In this section, we generalize the configuration context for an FL pipeline and define components and algorithms to perform adaptive orchestration of the pipeline based on the configuration context. Adaptive orchestration can be achieved by both responding to unexpected events and performing environment reconfigurations.

3.1 Architecture and Mechanisms

The proposed architecture for adaptive orchestration of FL workflows, shown in Figure 1, consists of two main components: orchestrator and node. This architecture is an extension of the general architecture for the orchestration of ML pipelines proposed in D1.1.

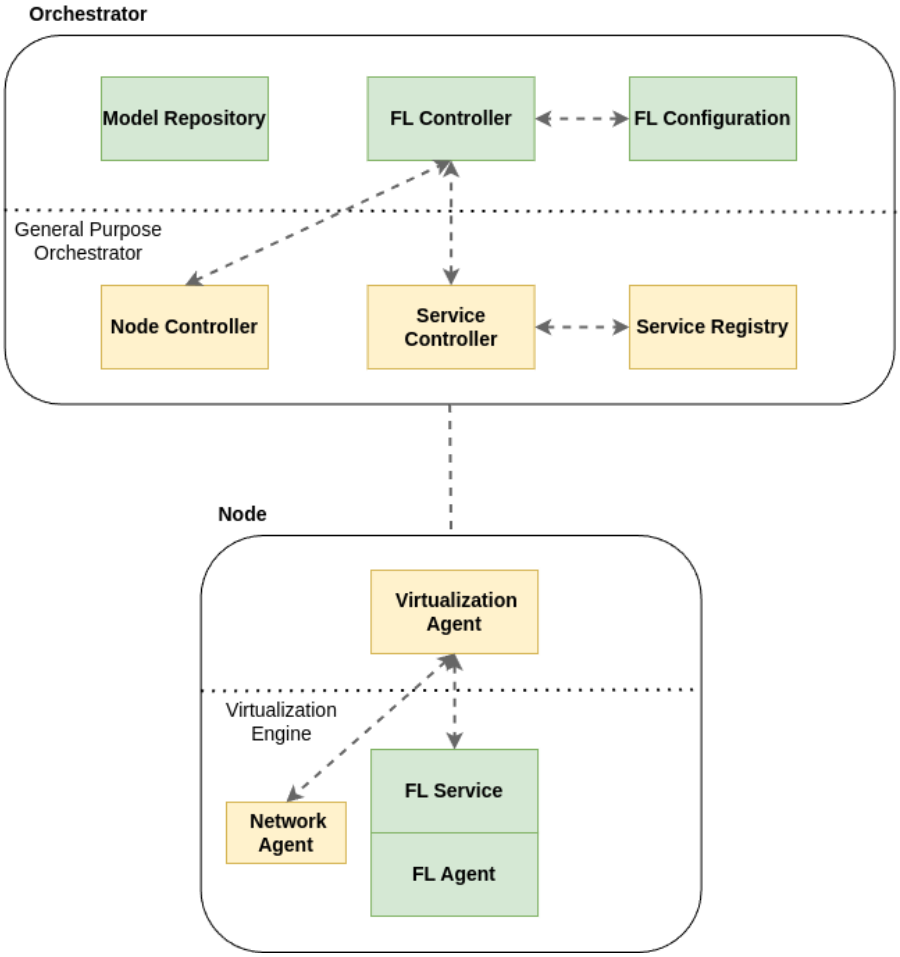


Figure 1: Architecture for adaptive orchestration of FL workflows.

The orchestrator's main role is to manage the FL pipeline. It extends the general-purpose orchestrator with the *FL Controller,* which is designed to control the FL pipeline, and the *FL Configuration,* which is a component in charge of obtaining the optimal pipeline configuration for a given environment. In addition, the architecture introduces *Model Repository,* a component that is used to store model snapshots during FL.

A node can be any device that can run an *FL Service* through a virtualization engine supported by the general-purpose orchestrator. An *FL Service* can be either an FL client or an aggregator, depending on the node's role. This service runs with a sidecar *FL Agent* that is designed to monitor an FL pipeline and to send FL reports to the *FL Controller.*

The workflow of the *FL Controller* can be divided into five phases:

1. Receiving training and cost configuration.
2. Collecting node features.
3. Obtaining optimal FL configuration.
4. Deploying FL components.
5. Monitoring the environment.

Orchestration is an iterative process, so the phases are repeated until the training target is reached. If the monitoring step detects an environmental change, the process is repeated starting from phase three. All the phases and the decisions made during runtime are shown in flowchart in Figure 2.

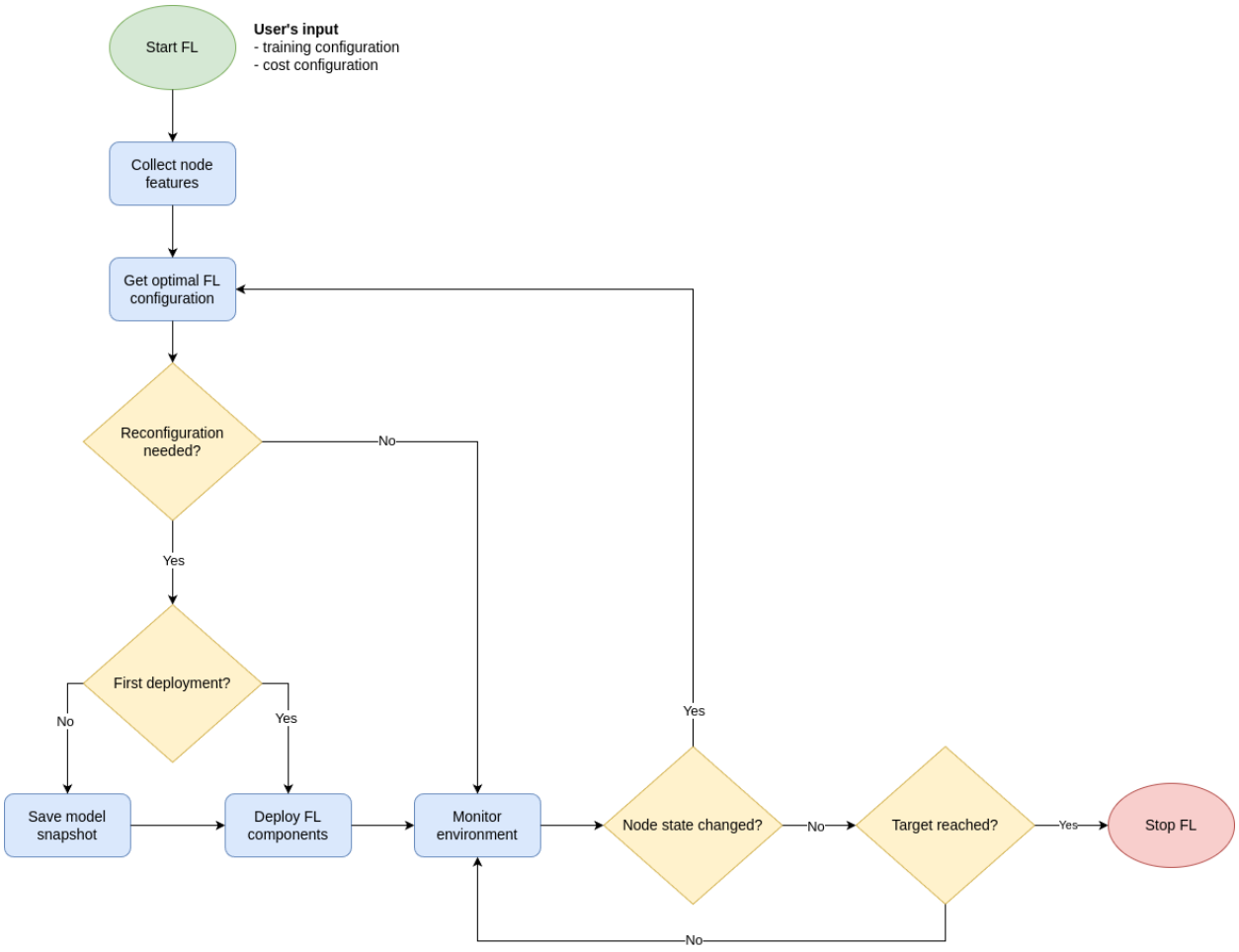


Figure 2: Flowchart of the FL controller's workflow.

The main goal of the orchestration is to provide autonomy and self-adaptiveness for our FL training system in runtime. To train a model, users only need to specify the training configuration and the cost for training. The whole pipeline can be executed automatically. The training configuration contains all the information needed to perform the model training: an ML model to be trained and training parameters such as batch size, learning rate, etc. Users can also specify cost for the training encoded in a cost configuration. The cost can be specified based on different factors, such as communication, computation, or time. The cost configuration is considered as the optimization goals for the orchestrator to perform the adaptivity, for example, to save the communication cost or to minimize the training time.

The next phase in the controller's workflow is to collect node features. This means discovering all nodes that could participate in FL and obtaining the following features:

- **Infrastructure-specific features.** First, the controller collects features related to the ECC infrastructure. These features are related to node computing resources and an underlying network that connects the nodes. In this step, the controller calculates the communication costs between all discovered nodes.
- **FL-specific features.** The next step is to obtain features that are related to FL. This step defines whether a node can take on the role of a client, meaning that it has the data available for training, or it can only be used for aggregation. For nodes with available training data, the controller extracts the data distribution with the help of an *FL Agent*. If the node already participated in FL, the historical information of its behavior can be extracted, such as training time, resource usage during training, etc.

Once all the inputs for orchestration are collected, the FL configuration can be obtained. All the inputs collected from the user and the node features are collected from the environment and passed to the FL configuration function. The architecture is designed so that it can incorporate any of the state-of-the-art configuration strategies, such as the ones proposed in [7][8][9]. The output of the configuration function defines the FL topology: clients, aggregators, and connections between them which are mapped to the ECC environment. Additionally, the configuration can output the number of local epochs and, in the case of hierarchical FL, the number of local aggregations before performing global aggregation.

To better understand how the configuration strategy works, we will give an example using the configuration strategy from [7]. In this work, the authors propose a hierarchical clustering mechanism for FL that performs a trade-off between communication cost and learning performance. Therefore, the user's input would be the model to be trained, training configuration, and cost configuration which would be defined to minimise the communication cost. The remaining inputs are obtained from the environment: nodes and their resources, data distribution on the nodes, and communication cost between the nodes. Once all the inputs are defined, the configuration function is run, and the configuration is obtained. It contains clients, aggregators, connections between them, the number of local epochs, and the number of local aggregations.

In phase four after the configuration is defined, the FL components are deployed, and an FL pipeline starts. In this phase, the client nodes download the model to be trained, connect to their parent aggregator defined in the configuration, and start training.

The adaptiveness of the orchestration is implemented in the fifth phase of the controller's workflow, which is monitoring the environment. Monitoring of the environment consists of:

- **Infrastructure monitoring:** Various infrastructure components that can have an impact on FL performance are monitored: node states and their resources, network state, etc.
- **FL performance monitoring:** FL controller can also monitor the actual FL performance to act if the performance is not satisfactory. Therefore, it can monitor changes in training accuracy, loss, time needed per epoch, etc.

If an unexpected event occurs during monitoring, such as a node leaving the system, a new configuration function is executed, i.e. controller's workflow is returned to phase three. However, reconfiguration of the environment may introduce new costs in terms of computation, communication, or time. Therefore, a decision can be made to skip a reconfiguration step with a new optimal configuration if the cost of reconfiguration is higher than the gain, and thus the FL pipeline would continue to run with a suboptimal configuration.

If a reconfiguration is needed, the controller repeats the deployment process from phase four. However, the current model state shouldn't be lost. Therefore, the global aggregator takes a snapshot of the latest global model and pushes it to the *Model Repository*. After that, the FL components are deployed; however, they do not start from the initial model defined by the user, but from the latest model snapshot. If reconfiguration isn't necessary, the controller continues to monitor the environment.

The FL pipeline is finished when the target is reached. This target is defined by the user and can be defined as: (i) reaching a target accuracy or loss, (ii) reaching cost limit (computation, communication, or time), or (iii) a predefined number of client epochs or global rounds.

3.2 Implementation: Adaptive FL Orchestration on Top of Kubernetes

The proposed architecture is implemented as an extension of Kubernetes, an industry-standard container orchestration tool, hence the implementation supports FL components running in containers.

3.2.1 FL Orchestrator

The FL orchestrator, consisting of *FL Controller, FL Configuration* and *Model Repository,* is built on top of Kubernetes. Therefore, in our implementation of the orchestrator, all yellow components of the architecture in Figure 1 are implemented by Kubernetes and used by the green, FL-specific components through Kubernetes API. It is important to emphasize that the integration with Kubernetes API doesn't limit the orchestration to using only Kubernetes, because the same API is implemented by its lightweight edge-aware distributions, such as K3s, KubeEdge, or microK8s.

The orchestrator's code is fully implemented in the Golang programming language, and it connects to the Kubernetes API to obtain infrastructure information and deploy FL components in containers. Figure 3 shows how the orchestrator is implemented on top of the Kubernetes tool.

Full implementation of the FL orchestrator is available at the following link:
https://github.com/AIoTwin/fl-orchestrator

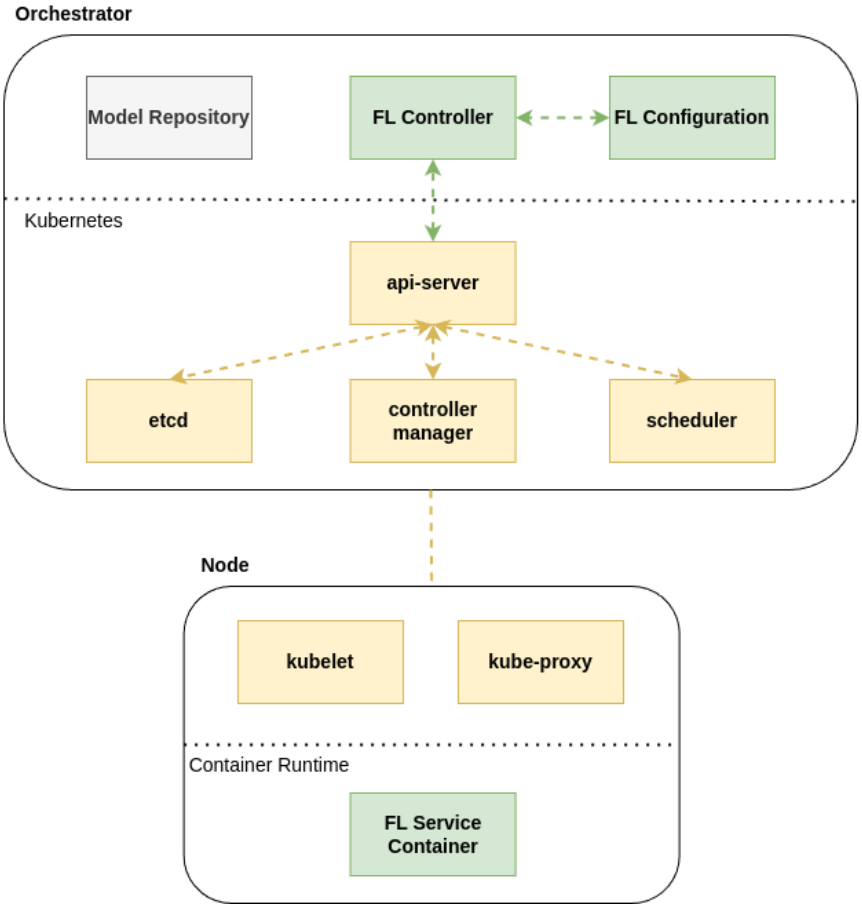


Figure 3: Implementation of the proposed architecture on top of Kubernetes.

The FL orchestrator currently implements the *FL Controller* and *FL Configuration*. The *Model Repository* will be added in the future version of the middleware. The FL controller's workflow phases from Section 3.1 are implemented as follows:

1. *Receiving training and cost configuration.* The controller receives a training configuration from a user through config files for model and aggregation. The cost configuration currently supports defining the communication budget per client epoch.
2. *Collecting node features.* The controller collects node features through the connection with the Kubernetes API. It collects available nodes and their resources through available API endpoints, as well as communication costs and data distribution that are stored in Kubernetes node labels.
3. *Obtaining optimal FL configuration.* The current FL configuration is implemented to support hierarchical FL. It obtains the optimal clusters in terms of data distribution within the communication budget defined by the user. In essence, it balances the KLD [1]between clusters in

[1] Kullback-Leibler Divergence (KLD) is a measure of how one probability distribution diverges from a second, expected probability distribution. In the context of clustering in hierarchical FL, KLD can be used to measure a difference in data distribution between two clusters.

a similar way as implemented in [7] but without the trade-off, as the communication budget is limited by the user.

4. *Deploying FL components.* FL components are deployed as containerized services. Each node can have one service that can serve as a client, local aggregator, or global aggregator, depending on the output of configuration. The implementation of the FL service and the configuration required for each service component will be described in the following section.
5. *Monitoring environment.* Current implementation supports monitoring node state changes. Therefore, the controller reacts to nodes joining and leaving the network, which is obtained from the subscriptions to the events in the Kubernetes API. If a node state change occurs, the reconfiguration is triggered, which removes all the FL components and deploys a new configuration.

To allow the simulation of different topologies and implement large scale experiments, we have also implemented the simulated version of the orchestrator. This version deploys the actual containers of FL services on a Kubernetes cluster but on a simulated topology. This allows performing large-scale experiments since multiple containers, that simulate nodes in FL, can be deployed on one node in the Kubernetes cluster. The topology is defined with a CSV file and the orchestration also allows introducing node state changes by editing the content of the file.

3.2.2 FL Service

The FL service of the proposed architecture can take on one of the three functionalities: client, local aggregator (LA), or global aggregator (GA). Therefore, we had to implement three services running in containers that can be externally configured to enable orchestration. We also implemented a framework for performing simulations of different FL configurations.

Full implementation of the framework and the FL service is available at the following link:
https://github.com/AIoTwin/fl-framework

The code of the FL service is built in the Python programming language. The dependencies are maintained in the file “requirements.txt“, a clear, short list of requirements maintained by Python's virtual environment (venv). First, a framework was implemented for performing simulations of different FL configurations on a single machine in the form of Python threads. After that, we extracted the code for each component (client, LA, GA) and implemented each component as a Docker container.

The FL service and the simulation framework are designed as an extension of Flower [10], a popular FL framework. We expanded Flower with an external configuration and the possibility of implementing a hierarchical architecture. Moreover, we extended this framework with a failure rate option that allows unreliable clients not to send updates in aggregation rounds based on a probability that can be selected for each client. The FL strategy that our framework currently supports is Federated Averaging (FedAvg); however, it could be extended with other strategies in the same way how Flower supports it [11].

The hierarchical architecture not only has clients and a global aggregator, but also local aggregators located between the clients and the global aggregator. In a hierarchical setup, clients send their trained models to the local aggregators after a specified number of epochs. Local aggregators then aggregate the received models and either send them back to the clients or, after a specified number of local aggregation

rounds, send the aggregated model to the global aggregator. Finally, the global aggregator aggregates the local models and send the updated global model back to local aggregators.

Simulations are configured through YAML files that are easily integrated into the system using the Spock framework [12]. These YAML files can be found and modified in the "config" directory. The simulations are designed to run on a single server, where users have the flexibility to choose between utilizing only CPU or also using GPU resources. This framework additionally offers the possibility of adding new model architectures and datasets for the tests. When logging is activated, the FL performance information is sent to Weights & Biases [13]. This implements the *FL Agent* of the proposed architecture that sends FL progress reports, and it is integrated into the same container as the *FL Service*.

Figure 4 shows an example of a hierarchical setup of the simulation that can be found in the file on path "config/example_hierarchical/entry_config.yaml".

```yaml
topology:
  root:
    base_config: &aggregator_config 'config/example_hierarchical/aggregator_config.yaml'
    children:
      aggregators:
        [ { base_config: *aggregator_config,
            children: {
              num_clients: 5,
              failure_rate: [ ],
              base_config: &client_config 'config/example_hierarchical/client_config.yaml'
            }
        },

          { base_config: *aggregator_config,
            children: {
              num_clients: 5,
              failure_rate: [ 0.6,0.7 ],
              base_config: *client_config
            }
          },
        ]
```

Figure 4: Example of a configuration for simulation of a hierarchical FL pipeline.

This example has two local aggregators, each with five clients. The number of clusters and clients can be changed as required. In the second cluster, there are two unreliable clients that have a failure probability of 60% and 70% meaning that they fail to deliver the model updates in 6 or 7 rounds out of 10. These failure rates can also be adjusted.

However, to enable orchestration, we had to separate the roles (client, LA, GA) into components and implement them as containers. Therefore, using the framework, we created Docker container images for the client, LA, and GA by extracting the framework implementation for the specific component and allowing external definition of the component configuration. All components are implemented on top of "python:3.10-slim", a lightweight container image with Python 3.10 runtime.

The client container, apart from model and dataset configuration, receives the following configuration parameters:

- **client_id**: ID of the client;
- **epochs**: number of local epochs; and
- **server_address**: address of its parent aggregator, which can be an LA or GA.

The most important configuration parameters for the local aggregator are the following:

- **num_clients:** number of clients attached to it;
- **rounds:** total number of rounds until finished;
- **local_rounds:** number of local aggregations before sending the model update to the global aggregator;
- **parent_address:** address of its parent aggregator; and
- **server_address:** the address where it is reachable.

The configuration of the global aggregator takes similar parameters but without the ones specific to the LA. These parameters contain:

- **num_clients:** number of clients attached to it (LAs are also considered as clients from the GA's perspective);
- **rounds:** total number of rounds until finished; and
- **server_address:** the address where it is reachable.

3.3 Evaluation

The evaluation of the implementation of adaptive orchestration of FL workflows and the proposed mechanisms was motivated by pilot experiments. These experiments were conducted as simulations using the framework detailed in Section 3.2.2. Following the simulations, we evaluated the orchestrator on a K3s cluster.

3.3.1 Pilot Experiments

The federated learning experiments were designed to identify how the training performance changes with a different number of clients, a different number of data samples per client, a different number of class labels per client, the distribution of clients to clusters according to their local data, and the change in the number of epochs and local aggregation rounds.

The pilot experiment tests were performed on a single server. This server was equipped with 32 CPUs, specifically the AMD Ryzen 9 5950X 16-Core Processor, each with a maximum clock speed of 5083.3979 MHz. Additionally, the server incorporated 2 NVIDIA GeForce RTX 3090 GPUs, each featuring 24265 MiB of memory.

For the following image classification experiments, we used a convolutional neural network (CNN) architecture implemented in PyTorch. The architecture of CNN is shown in Figure 5 and consists of two convolutional layers with ReLU activation and max pooling followed by two fully connected layers. The network takes RGB images as input and outputs class labels. The dataset used for the experiments was CIFAR-10 [14], which contains 60,000 32x32 colour photos with 10 different classifications. The ten distinct classes stand for vehicles, trucks, cars, frogs, dogs, cats, deer, and animals. Each class has 6,000 pictures.

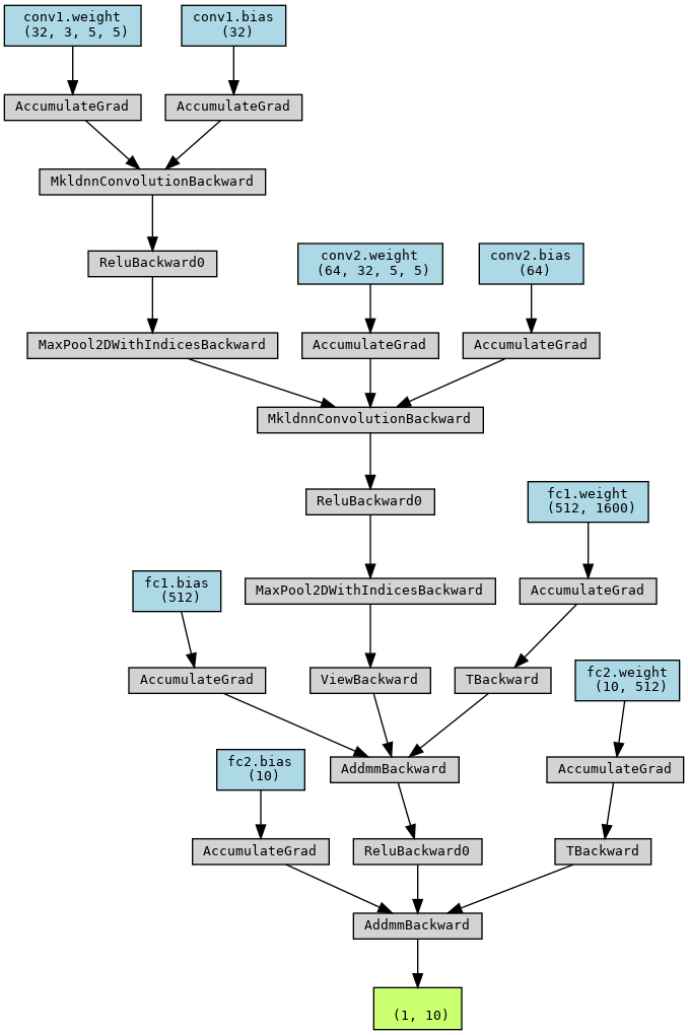


Figure 5: Architecture of CNN used in pilot experiments.

The following three graphics show the different data distributions when 20 clients share the same number of data samples. The samples that belong to one class label are depicted in one colour. The x-axis represents the client ID, and the y-axis represents the number of samples. When looking at the client c1, one can observe that in the first strategy in Figure 6, this client has only samples from one class; in Figure 7, this client has the same amount of data samples but from two different classes. In Figure 8, the client has samples from all classes.

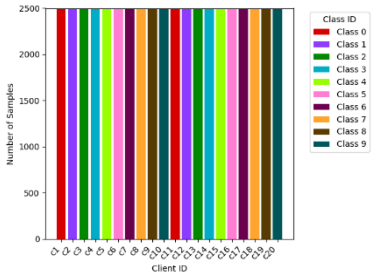


Figure 6: Strategy of 1 label per client.

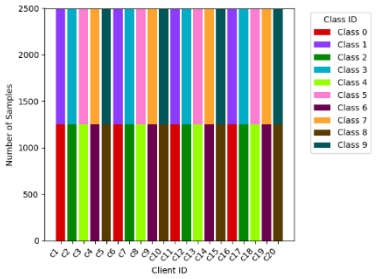


Figure 7: Strategy of 2 labels per client.

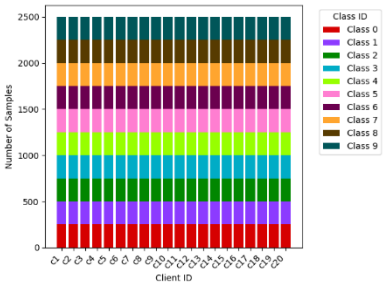


Figure 8: Strategy of 10 labels per client.

Regarding the data distribution, the highest accuracy was achieved when the data was balanced between clients. Table 1 compares the training accuracy for different data distributions of 20 clients when the data is split into 20 equal parts. The clients directly send their updates to the global aggregator for the following results, and the accuracy was measured after 500 training epochs. It can be observed that the best results were achieved when clients had more labels to train with locally.

Table 1: Comparison of accuracy with different data distributions.

Strategy	Accuracy after 500 epochs
1 label/client	22.71
2 labels/client	47.92
10 labels/client	69.39

Table 2 depicts the accuracy after 500 training epochs in a scenario when each client holds samples from two labels. In this table, we can see comparison in accuracy for different numbers of clients and different amounts of data. For a better overview, this table compares the configurations without LAs. The experiments show that better accuracy can be achieved when clients have more data. When 10 clients are included and the number of splits is 10, each client has 1/10 of the dataset. Interestingly, the experiments show better results when having fewer clients, even if they have less data.

Table 2: Comparison of accuracy with different data splits and number of clients.

Number of clients	Number of data splits	Accuracy after 500 epochs
10	10	57.54
10	15	54.84
10	30	48.98
10	60	48.88
20	20	47.92

In the next part of the pilot experiments, we compared non-IID and IID clusters. Two hierarchical configurations used in the experiments are shown in Figure 9.

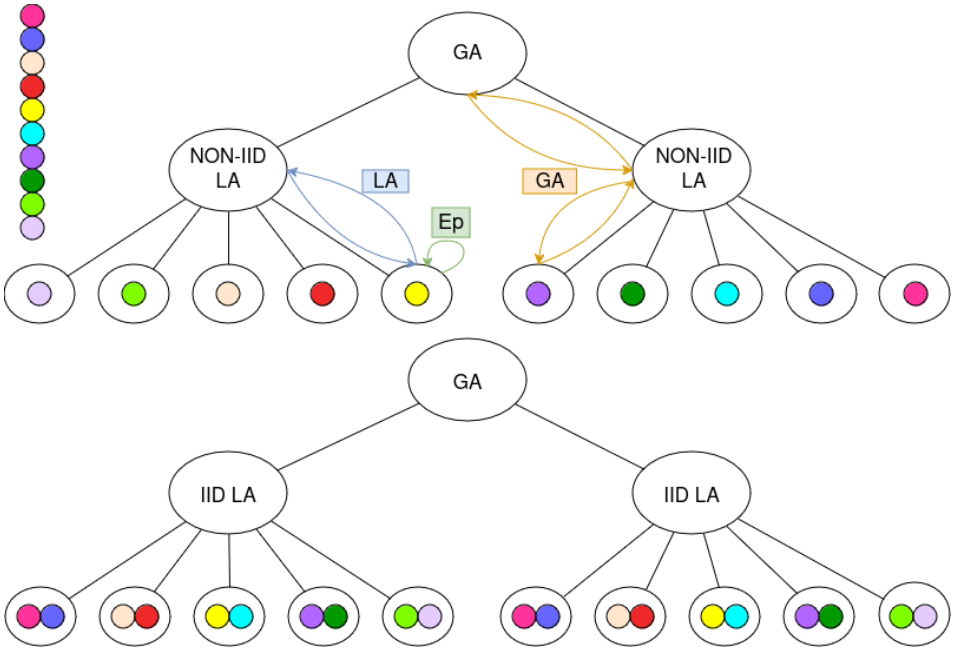


Figure 9: Clustering strategies for hierarchical FL.

Each of these structures has a global aggregator on the top, two local aggregators below, and each of the local aggregators has 5 clients. The differently coloured points in the clients represent samples from a specific label. The non-IID local aggregators only have samples from half of the classes (first configuration), whereas the IID LA have samples from all classes (second configuration).

In this figure, we can also see the difference between epochs (Ep), local aggregations (LA) and global aggregations (GA). Epochs are the number of rounds that one client trains for before the trained model is sent to the global aggregator. Local aggregations are the number of aggregations at a local aggregator before a global aggregation.

Table 3 shows the accuracy results of two different configurations. For these experiments, we used 20 clients and split the data into 20 parts.

Table 3: Comparison of different epochs/local aggregation strategies and IID/non-IID clusters.

IID cluster	Hierarchical	Ep	LA	Accuracy after 500 epochs
No	No	10	-	22.73
No	Yes	10	1	22.59
No	Yes	5	2	24.33
No	Yes	2	5	22.91
Yes	No	10	-	22.71
Yes	Yes	5	2	22.73
Yes	Yes	2	5	21.37

For the non-IID cluster, the hierarchical structure outperformed the structure with no LAs in almost all cases. Interestingly, in these experiments, the hierarchical structure with 5 local epochs and 2 local rounds performed better than the one with higher aggregation frequency. It is also interesting that the IID clusters performed worse in our experiments than the non-IID clusters.

3.3.2 Deployment in a K3s Cluster

The implementation of the orchestrator was evaluated on a K3s cluster. As it was mentioned in Section 3.2.1, the orchestrator is implemented on top of Kubernetes API, so it works with all its distributions. Based on the comparison in [15] and the analysis in deliverable D1.2, we decided to deploy a K3s [16] cluster as it is lightweight, edge-aware, and provides simple deployment and maintenance. Figure 10 shows the configuration of the K3s cluster.

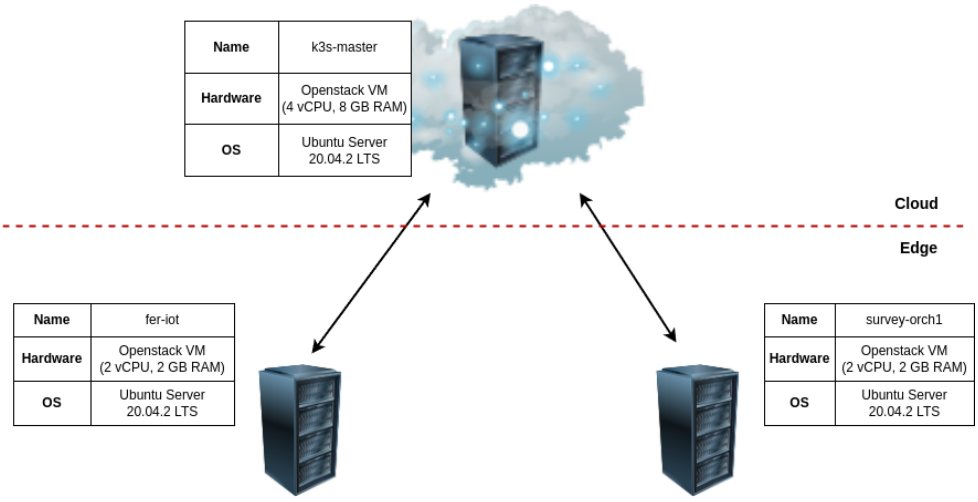


Figure 10: K3s cluster configuration.

The cluster deployed for evaluation consists of three nodes, each of them running on OpenStack VMs. One node was serving as the orchestrator of the cluster and two were worker nodes.

3.3.2.1 Simulated Infrastructure

The first orchestrator evaluation was performed on a simulated infrastructure. The reason for that was to test different hierarchical configurations and environmental changes which would be difficult to implement in the current cluster setup of three nodes. Therefore, we created a simulated infrastructure through a CSV file as it was mentioned in Section 3.2.1. The infrastructure definition is shown in Figure 11.

```
n1,aggregator,"n2:100,n3:100",""
n2,aggregator,"n1:100,n3:200",""
n3,aggregator,"n1:100,n2:200",""
n4,client,"n1:200,n2:100,n3:50","0:1000,1:10000,2:20000"
n5,client,"n1:200,n2:100,n3:100","3:1000,4:10000,5:20000"
n6,client,"n1:200,n2:100,n3:100","6:1000,7:10000,8:20000"
n7,client,"n1:200,n2:50,n3:100","0:1000,4:10000,9:20000"
n8,client,"n1:200,n2:100,n3:100","0:1000,4:10000,9:20000"
n9,client,"n1:200,n2:100,n3:50","1:1000,5:10000,7:20000"
```

Figure 11: Infrastructure definition CSV file.

Infrastructure is defined by specifying each node with its role, communication costs towards the aggregators, and data distribution on the node. In this setup, there are three aggregator nodes and six client nodes. Client nodes each hold samples of random three classes of the CIFAR-10 dataset. In the experiment, clients were training a CNN detailed in Section 3.3.1 over the CIFAR-10 dataset.

As it was described in Section 3.2.1, current implementation of the orchestrator obtains the optimal clusters in terms of data distribution within the communication budget defined by the user. First, the configuration function defines clusters with the closest distribution to the average distribution of the whole dataset, i.e. all data available on clients. Then, it defines the number of epochs and local rounds with the lowest value that satisfies the communication constraints.

FL orchestration was run with the model size set to 10 MB and the communication budget set to 600 units per epoch which served just a demonstrative example. Figure 12 shows the output of the initial FL configuration and Figure 13 shows that the FL services were actually deployed ("ga" in the name stands for the global aggregator, "la" for a local aggregator, and "cl" for a client). It can be observed that multiple services are running on the same node as it is the simulated experiment.

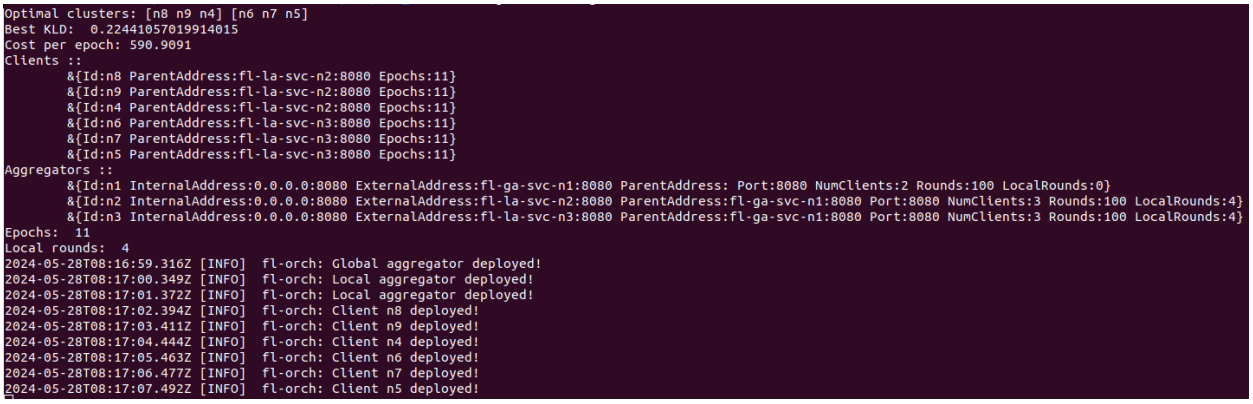


Figure 12: Initial configuration and deployment of FL components (simulated infrastructure).

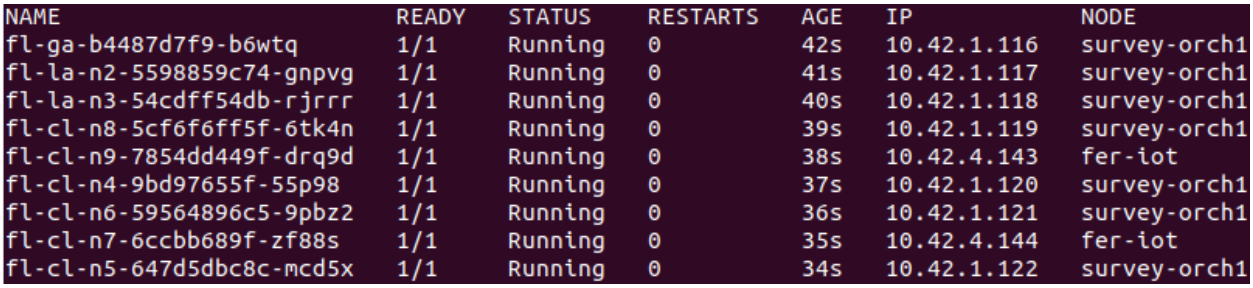


Figure 13: Pods running in the cluster (simulated infrastructure).

Figure 14 shows the output of FL configuration: clients, aggregators, epochs and local rounds. The number of epochs and local rounds is calculated to be the lowest number that meets the communication constraints. Considering that the model size is 10 MB, communication budget is 600 units per epoch, the communication costs as were defined in Figure 14, we can calculate the communication cost per local epoch as follows:

$$localAggrCost = localAggrA + localAggrB = 6000\,unit$$

$$commCostPerEpoch = \frac{localAggrCost * localRounds + globalAggrCost}{localRounds * epochs}$$

$$commCostPerEpoch = \frac{6000\,unit * 4 + 2000\,unit}{44\,epoch} = \mathbf{590.91}\,\frac{\boldsymbol{unit}}{\boldsymbol{epoch}}$$

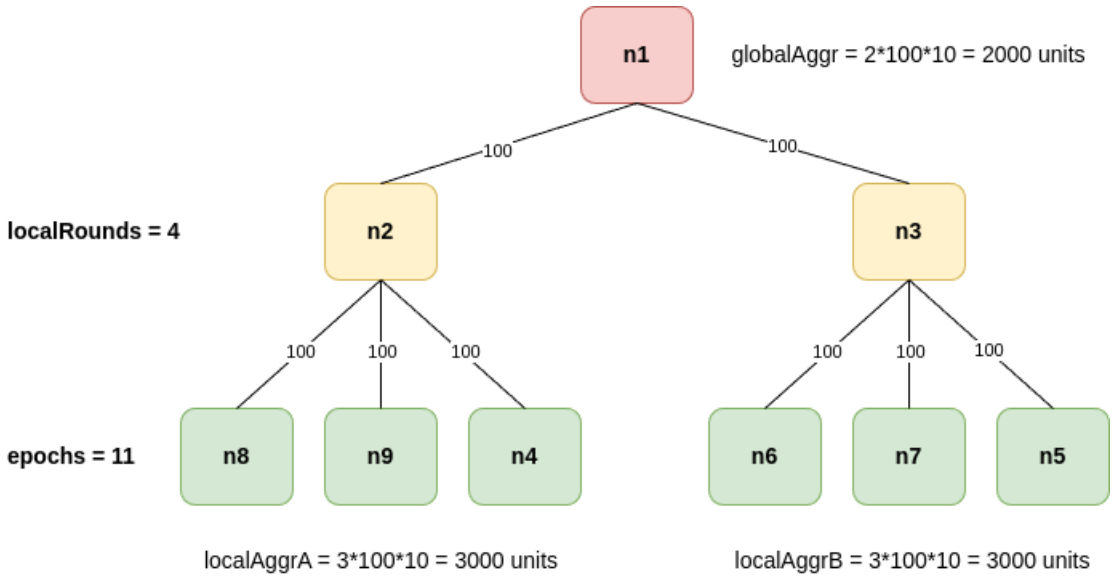


Figure 14: Output of FL configuration for experiment with simulated infrastructure.

During the FL runtime, we removed node "n9" from the cluster. Figure 15 shows how removing a node from the cluster was handled by the orchestrator. The orchestrator detected the change, ran a new configuration strategy and reconfigured the FL pipeline with new clusters without the removed node. In addition, since removing a node will decrease the communication cost per epoch as we have lower number of model updates sent through the network, the aggregation frequency was increased by lowering the number of epochs while still meeting the communication constraints (9 epochs and 4 local rounds).

```
New event:
Nodes added: []
Node removed: [0xc0004f7f20]
Removed previous deployment and reconfiguring...
Optimal clusters: [n5 n4 n7] [n6 n8]
Best KLD:  0.37967447155304745
Cost per epoch: 555.55554
Clients ::
        &{Id:n5 ParentAddress:fl-la-svc-n2:8080 Epochs:9}
        &{Id:n4 ParentAddress:fl-la-svc-n2:8080 Epochs:9}
        &{Id:n7 ParentAddress:fl-la-svc-n2:8080 Epochs:9}
        &{Id:n6 ParentAddress:fl-la-svc-n3:8080 Epochs:9}
        &{Id:n8 ParentAddress:fl-la-svc-n3:8080 Epochs:9}
Aggregators ::
        &{Id:n1 InternalAddress:0.0.0.0:8080 ExternalAddress:fl-ga-svc-n1:8080 ParentAddress: Port:8080 NumClients:2 Rounds:100 LocalRounds:0}
        &{Id:n2 InternalAddress:0.0.0.0:8080 ExternalAddress:fl-la-svc-n2:8080 ParentAddress:fl-ga-svc-n1:8080 Port:8080 NumClients:3 Rounds:100 LocalRounds:4}
        &{Id:n3 InternalAddress:0.0.0.0:8080 ExternalAddress:fl-la-svc-n3:8080 ParentAddress:fl-ga-svc-n1:8080 Port:8080 NumClients:2 Rounds:100 LocalRounds:4}
Epochs:  9
Local rounds:  4
2024-05-28T08:18:51.454Z [INFO]  fl-orch: Global aggregator deployed!
2024-05-28T08:18:52.484Z [INFO]  fl-orch: Local aggregator deployed!
2024-05-28T08:18:53.505Z [INFO]  fl-orch: Local aggregator deployed!
2024-05-28T08:18:54.515Z [INFO]  fl-orch: Client n5 deployed!
2024-05-28T08:18:55.525Z [INFO]  fl-orch: Client n4 deployed!
2024-05-28T08:18:56.538Z [INFO]  fl-orch: Client n7 deployed!
2024-05-28T08:18:57.552Z [INFO]  fl-orch: Client n6 deployed!
2024-05-28T08:18:58.561Z [INFO]  fl-orch: Client n8 deployed!
```

Figure 15: Node removed event handling.

3.3.2.2 Real Infrastructure

The evaluation was also performed on the real infrastructure, meaning that one cluster node can only host one FL service, like in a real-world deployment. Due to the small scale of the test cluster, we could only evaluate the centralized FL. Figure 16 shows the nodes available in K3s cluster and their labels that hold the information on the communication costs and data distribution. Node “k3s-master” had the role of the aggregator and nodes “fer-iot” and “survey-orch1” were client nodes.

```
NAME           STATUS   ROLES                  AGE   VERSION        LABELS
fer-iot        Ready    <none>                 69d   v1.28.7+k3s1   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=k3s,beta.kubernetes.io/os=linux,com
m/k3s-master=100,data/2=10000,data/3=20000,fl/type=client,kubernetes.io/arch=amd64,kubernetes.io/hostname=fer-iot,kubernetes.io/os=linux,node.kubernetes.io/instance-t
ype=k3s
survey-orch1   Ready    <none>                 79d   v1.28.6+k3s2   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=k3s,beta.kubernetes.io/os=linux,com
m/k3s-master=100,data/0=100,data/1=1000,fl/type=client,kubernetes.io/arch=amd64,kubernetes.io/hostname=survey-orch1,kubernetes.io/os=linux,node.kubernetes.io/instance
-type=k3s
k3s-master     Ready    control-plane,master   79d   v1.28.6+k3s2   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=k3s,beta.kubernetes.io/os=linux,fl/
type=aggregator,kubernetes.io/arch=amd64,kubernetes.io/hostname=k3s-master,kubernetes.io/os=linux,node-role.kubernetes.io/control-plane=true,node-role.kubernetes.io/m
aster=true,node.kubernetes.io/instance-type=k3s
```

Figure 16: K3s nodes and their labels.

This evaluation was also training a CNN over the CIFAR-10 dataset. FL orchestration was also run with the model size set to 10 MB and the communication budget set to 600 units per epoch. Figure 17 shows the output of the initial FL configuration and Figure 18 shows that the FL services were actually deployed and were running on separate nodes. The number of epochs is calculated to be the lowest number that meets the communication constraints. Considering that the model size is 10 MB, communication budget is 600 units per epoch, the communication cost on the link from “survey-orch” to the aggregator is 100 units per MB and cost on the link from “fer-iot” to the aggregator is also 100 units per MB, we can calculate the communication cost per local epoch as follows:

$$commCostPerEpoch = \frac{100\ \frac{unit}{MB} * 10\ MB + 100 \frac{unit}{MB} * 10\ MB}{4\ epoch} = \frac{2000\ unit}{4\ epoch} = \mathbf{500} \frac{\boldsymbol{unit}}{\boldsymbol{epoch}}$$

```
Clients ::
        &{Id:fer-iot ParentAddress:fl-ga-svc-k3s-master:8080 Epochs:4}
        &{Id:survey-orch1 ParentAddress:fl-ga-svc-k3s-master:8080 Epochs:4}
Aggregators ::
        &{Id:k3s-master InternalAddress:0.0.0.0:8080 ExternalAddress:fl-ga-svc-k3s-master:8080 ParentAddress: Port:8080 NumClients:2 Rounds:100 LocalRounds:0}
Epochs:  4
Local rounds:  0
2024-05-14T21:28:33.993Z [INFO]  fl-orch: Global aggregator deployed!
2024-05-14T21:28:35.017Z [INFO]  fl-orch: Client fer-iot deployed!
2024-05-14T21:28:36.029Z [INFO]  fl-orch: Client survey-orch1 deployed!
```

Figure 17: Initial configuration and deployment of FL components (real infrastructure).

```
NAME                                  READY   STATUS    RESTARTS   AGE   IP             NODE
fl-ga-7c9c58cf9d-n5lwm                1/1     Running   0          40s   10.42.0.104    k3s-master
fl-cl-fer-iot-67584d889c-clnth        1/1     Running   0          39s   10.42.4.115    fer-iot
fl-cl-survey-orch1-798b78ff6f-rxh5s   1/1     Running   0          38s   10.42.1.49     survey-orch1
```

Figure 18: Pods running in the cluster (real infrastructure).

During the FL runtime, we added a new node “cilic-f1-1” to the cluster. Figure 19 shows how adding a new node to the cluster was handled by the orchestrator. The orchestrator detected the change, ran a new configuration strategy and reconfigured the FL pipeline with the new node serving as client. In addition, since the new node sends model updates which will increase the communication cost per epoch, the aggregation frequency was reduced by increasing the number of local epochs to stay within the communication budget.

```
New event:
Nodes added: [0xc000410cc0]
Node removed: []
Removed previous deployment and reconfiguring...
Clients ::
        &{Id:fer-iot ParentAddress:fl-ga-svc-k3s-master:8080 Epochs:5}
        &{Id:survey-orch1 ParentAddress:fl-ga-svc-k3s-master:8080 Epochs:5}
        &{Id:cilic-fl-1 ParentAddress:fl-ga-svc-k3s-master:8080 Epochs:5}
Aggregators ::
        &{Id:k3s-master InternalAddress:0.0.0.0:8080 ExternalAddress:fl-ga-svc-k3s-master:8080 ParentAddress: Port:8080 NumClients:3 Rounds:100 LocalRounds:0}
Epochs:  5
Local rounds:  0
2024-05-14T21:33:06.102Z [INFO]  fl-orch: Global aggregator deployed!
2024-05-14T21:33:07.117Z [INFO]  fl-orch: Client fer-iot deployed!
2024-05-14T21:33:08.138Z [INFO]  fl-orch: Client survey-orch1 deployed!
2024-05-14T21:33:09.157Z [INFO]  fl-orch: Client cilic-fl-1 deployed!
```

Figure 19: New node event handling.

4 Inference Pipeline: QoS-aware Data Routing in ECC

In a highly dynamic edge-to-cloud environment characterized by continuous changes in the states and locations of IoT devices and edge nodes, the services running on the edge nodes have to be scheduled, deployed and managed to meet potentially diverse Quality of Service (QoS) objectives, such as high availability or low response times. A significant body of works [17]-[24] on QoS-aware service orchestration in edge-cloud environments focuses on optimal service placement on compute nodes, often considering the volatility of the environment typically experienced at the edge. Once services are deployed across the computing continuum, the challenge arises on how to ensure continuous data delivery from IoT devices to the corresponding distributed service instances, while taking into account the dynamic nature of the edge-to-cloud environment, i.e., the ECC.

In practice, each service is associated to a set of QoS requirements specifying Service Level Objectives (SLOs) that must be met for all clients using the service. These requirements are based on various parameters, including latency, throughput, and security. To enable adaptive QoS-aware data routing in ECC while minimizing the load on resource-limited IoT devices, we introduce a new component named QEdgeProxy[2]: a QoS-aware load balancer designed as a QoS agent for IoT clients. Its role is to identify service instances across the continuum that could fulfil the QoS requirements for a given service, forming a ``QoS pool'', and execute load balancing among them. What sets our approach apart from related works, such as [25][26][27], is the design of the proxy to adaptively and consistently fulfil QoS requirements for each client based on the current state of the environment, rather than identifying instances that maximize QoS or perform trade-offs between QoS and load balancing. With this technique, we can achieve load balancing while ensuring that clients receive satisfactory QoS.

QoS-aware data routing is important in the context of the AIoTwin middleware to support routing towards the inference services. Once the inference services are deployed across the ECC, a mechanism is needed to route the inference requests to an appropriate service while making sure that the inference request meets the QoS requirements for the service. These requirements can be network-specific, such as latency or throughput, or service-specific, such as the minimal accuracy of inference. The requirements can also be defined in multiple dimensions. For example, an inference task usually requires high computing resources, so a lightweight edge node, which is usually available at the far edge of ECC close to the IoT devices that require inference services, might need to host an inference service with a lightweight model that provides faster inference but with lowered accuracy. Hence, a requirement can be defined like this: the average latency of processing a service request must be below 50 milliseconds and 80% of service requests must be served by the service with inference accuracy over 95%.

[2] The following paper has been accepted for publication at IEEE EDGE 2024 which presents QEdgeProxy: Ivan Čilić, Valentin Jukanović, Ivana Podnar Žarko, Pantelis Frangoudis, Schahram Dustdar “QEdgeProxy: QoS-Aware Load Balancing for IoT Services in the Computing Continuum.”

4.1 Architecture and Mechanisms

The core functionalities of the QEdgeProxy include (i) dynamically maintaining a set of service instances that meet the targeted QoS for a given service, and (ii) forwarding service requests to these instances while performing load balancing. By placing the proxy as an intermediary between the clients and IoT services across ECC, we remove the responsibility of the client to discover and select optimal service instances to connect with. Given that QoS is heavily influenced by the processing node where an instance is deployed, and considering the varying QoS requirements of different services, this approach broadens the pool of available nodes for request processing, potentially enhancing the efficiency of load balancing mechanisms.

As shown in Figure 20, QEdgeProxy is a component deployed on every node within the ECC. It integrates seamlessly into existing edge solutions, which are typically managed by a central *orchestrator* component, as in existing service orchestration tools such as Kubernetes. Services deployed across the heterogeneous ECC environment are usually packed in containers in order to achieve portability and fast startup times which is needed in this dynamic environment.

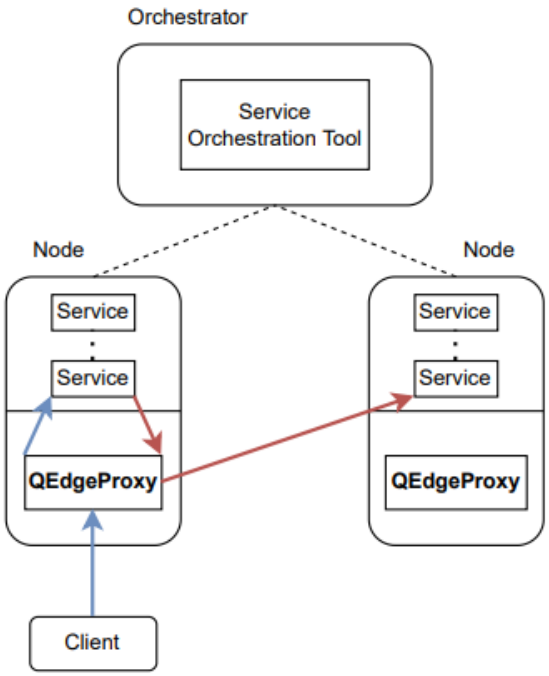


Figure 20: QEdgeProxy within the ECC.

The QEdgeProxy is designed to be exposed on each node in the ECC and to run just as any other service using the service orchestration tool. Clients initiating requests to IoT services must connect to its dedicated QEdgeProxy to be able to access that service. This proxy then forwards the request to the appropriate service instance. As shown in Figure 20, the proxy enables both client-to-service communication (blue arrows) and service-to-service communication (red arrows). Internal service clients, such as services running within the ECC that use other services in the continuum, route their requests through the QEdgeProxy instance running on their node. External clients, primarily IoT devices operating outside the ECC cluster, connect to the QEdgeProxy running on a node in their proximity.

The overall functionality of a QEdgeProxy can be summarized as the process of constructing and maintaining a *QoS pool* for a given service, i.e. a set of service instances capable of fulfilling the specified QoS requirements. This pool serves as the basis for routing incoming requests to the most suitable instances, ensuring uninterrupted data delivery and adherence to QoS constraints. The functionality of the QEdgeProxy is thus defined by two main algorithms: the routing algorithm and the monitoring algorithm.

```
1: procedure REQUESTROUTING(request, service)
2:     serviceInstance ← getInstanceFromQosPool(service)
3:     qos ← sendRequest(request, serviceInstance)
4:     storeQosMeasurements(serviceInstance, qos)
5: end procedure
```

Figure 21: Request routing algorithm.

Algorithm in Figure 21 defines the high-level request routing logic executed by a QEdgeProxy. When a request arrives for a target service, the QEdgeProxy selects an instance from the previously built QoS pool and forwards the request to the selected instance. After routing the request, the proxy measures the QoS attained and stores the observed values so that this information can be used for maintaining the QoS pool.

```
1: procedure MONITORING(event)
2:    if event == firstServiceRequest then
3:       initQosPool(event.service)
4:    else if event = instanceStateChange then
5:       updateQosPool(event.service,event.instance)
6:    else if event = newQosMeasurement then
7:       updateQosPool(event.service,event.measurement)
8:    else if event = environmentChange then
9:       updateAllQosPools(event.change)
10:    end if
11: end procedure
```

Figure 22: Monitoring algorithm.

To maintain the QoS pool up-to-date, the QEdgeProxy performs event-based system monitoring described in algorithm in Figure 22. There are four events that can trigger actions upon which QoS pools are updated:

- **Initial QoS Pool Creation:** Upon receiving the first request for a given service, the proxy establishes a new dedicated QoS pool based on initial estimates, such as environmental conditions or historical interactions with instances of similar services running on the same nodes.
- **Instance State Updates:** Through the connection with the orchestrator, the proxy continuously monitors the state of service instances, including instances joining or leaving the network. Upon detecting any instance state changes, the proxy updates the relevant QoS pool accordingly.
- **QoS Measurement Updates:** After forwarding a request to a service instance and measuring its QoS, the proxy incorporates these measurements into the corresponding QoS pool, ensuring that the pool reflects the actual performance of the instances.

- **Environment Monitoring and Reactive Updates:** The QEdgeProxy continuously monitors the environment for any significant changes, such as network disruptions or computing resource fluctuations on ECC nodes. Upon detecting such changes, the proxy triggers a comprehensive evaluation of all QoS pools affected by the environmental alterations and updates them accordingly to maintain QoS consistency.

Each service is associated with a set of minimum QoS requirements that define the acceptable performance levels for the service to meet its intended purpose for any client. These requirements a related to service performance and may include:

- **Latency**: Request latency, a crucial QoS metric, specifies the maximum allowable delay between submitting a request and receiving a response.
- **Throughput**: The target service's ability to handle a specific request rate is typically expressed in requests per second.
- **Reliability**: Reliability requirements define the percentage of requests that must be processed successfully, ensuring data integrity and minimizing the occurrence of errors.
- **Availability**: Availability is defined as the probability that a service is operational at any given point in time, allowing uninterrupted data delivery and service access.
- **Security**: Security requirements encompass various aspects, including protocol support, hardware specifications of the service-hosting node, and privacy measures such as limiting the number of network hops a request can traverse.
- **Service-specific requirements**: Each service domain can have its own requirements, such as the accuracy of inference described at the beginning of this section.

Each service consists of multiple replicas, referred to as instances. Each QEdgeProxy predicts the QoS of a specific instance based on approximations and its past behaviour, e.g., measured network latency based on past interactions with an instance. Various techniques can be utilized for predicting QoS based on recent QoS measurements and interactions with the node running the instance, e.g. LSTMs, reinforcement learning, or other ML techniques. Finally, the *QoS pool* is a subset of service instances that are likely to satisfy the QoS requirements.

4.2 Implementation: QEdgeProxy within Kubernetes

In this section, we provide the QEdgeProxy implementation within the Kubernetes ecosystem. As demonstrated in Deliverable D1.2, Kubernetes is a powerful and stable container orchestration platform and the integration can be used with its lightweight distributions, such as K3s [16], which is well-suited for edge computing environments.

Full implementation of the QEdgeProxy is available at the following link:
https://github.com/AIoTwin/qedgeproxy

Several built-in and open-source mechanisms in Kubernetes can be used to implement adaptive and custom routing at the edge. One of them is the NodePort service type which exposes a dedicated port on each node in the cluster to access a particular service: thus, an end device can connect to any node in the ECC to send a request to this service. However, behind the NodePort service, kube-proxy equally balances requests to the target pods, without any possibility to implement custom parameters, such as selecting nearby pods. Another Kubernetes feature, the NetworkPolicy resource, enables defining rules for communication between pods. For instance, setting *externalTrafficPolicy: Local* specifies that external traffic should be processed locally on the node that receives it. However, this solution only allows processing a request on the node which received it, and if no adequate pod is found, the request is discarded. On the other hand, Istio [28] is an open service management platform that implements the service mesh architectural pattern, simplifying cluster traffic and security management. It works by injecting a specific container into pods, known as a "sidecar proxy", which handles communication with other services and allows for defining additional traffic routing rules, security measures, and visibility. While Istio supports routing settings considering the locality of individual pods, it differs from adaptive routing, as it requires manual setting of regions and zones for resources. This approach contradicts the principle of adaptive routing, making Istio unsuitable for implementing the QEdgeProxy.

However, none of the proposed approaches provides the possibility to implement custom routing rules with requirements from our architecture, so we decided to implement a custom external component for adaptive request routing and deploy it in Kubernetes as a DaemonSet. DaemonSet is a Kubernetes resource which ensures that all worker nodes run a copy of a given pod, which in our case is the QEdgeProxy.

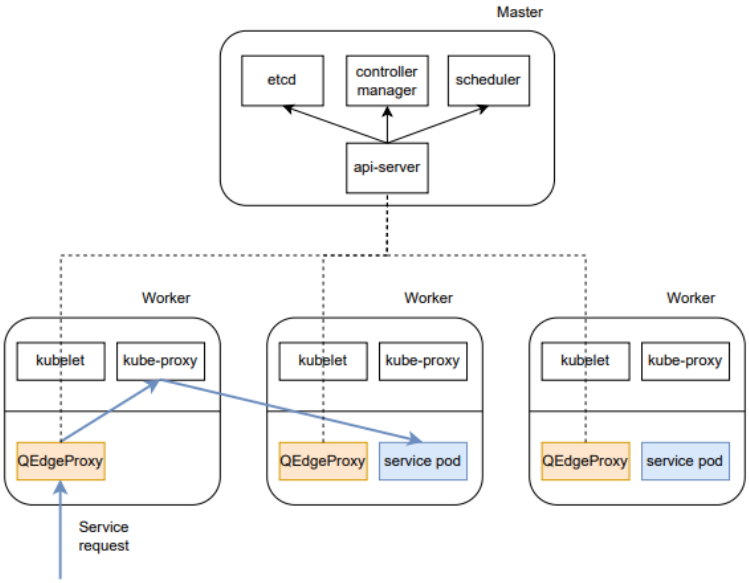


Figure 23: QEdgeProxy within a Kubernetes environment.

QEdgeProxy is seamlessly deployed within a Kubernetes environment, just like any other pod, as depicted in Figure 23. It is an HTTP server written in Golang, establishing connectivity with the Kubernetes API server to retrieve information on service instances across the ECC and the hosting nodes. By subscribing to events within the Kubernetes API, QEdgeProxy continuously monitors the status of the ECC. Clients access services by sending their HTTP requests to a designated QEdgeProxy, with the service name specified in the request header. Upon selection of a service instance by QEdgeProxy, the request is routed to the corresponding pod of the chosen service instance through “kube-proxy”.

4.3 Evaluation

To evaluate the effectiveness of our QEdgeProxy implementation, we conducted a comparative experimental evaluation using three different configurations on a K3s cluster, a lightweight distribution of Kubernetes. These configurations differ in the request routing mechanism employed, while maintaining a consistent execution flow. To align with the state-of-the-art and compare our solution against relevant approaches, we focused on exploring latency as a single QoS requirement in this evaluation, although our QEdgeProxy can support a more complex QoS specification. The configurations used in experiments are outlined below:

- **Kubernetes NodePort Service**: This configuration uses the built-in Kubernetes NodePort Service which exposes a port on every node, enabling direct access to IoT services (as described in Section 4.2). This comparison provides insights into the QEdgeProxy's performance relative to a standard Kubernetes routing solution.
- **Proximity-based routing**: This configuration implements the proximity-based routing algorithm *proxy-mity*, proposed by Fahs and Pierre in [26]. This algorithm incorporates a key parameter α that controls the trade-off between load balancing (when α = 0.0) and pure proximity-based routing (when α = 1.0). Two sub-configurations were evaluated: (i) *proxy-mity 1.0* with α = 1.0, prioritizing proximity for minimal latency; and (ii) *proxy-mity 0.8* with α = 0.8, achieving a balance between proximity and load balancing. The reason for choosing these values in two sub-configurations is that we wanted to compare our solution first with the configuration that favors proximity and offers the lowest latency, and second with another configuration which favors proximity with a slight trade-off with load balancing, which we believe is the most similar to the QEdgeProxy behavior.
- **QEdgeProxy**: This configuration employs our QEdgeProxy as the primary request routing mechanism from IoT clients to IoT services. As described in Section 4.2, a QEdgeProxy instance is deployed as a Kubernetes DaemonSet and runs on every node in the cluster, enabling dynamic routing and load balancing to meet the predefined service QoS requirements.

4.3.1 Experimental Setup

The K3s cluster in which the experiments are conducted consists of 7 nodes, six worker nodes, and one master node. All nodes are implemented as virtual machines using VMWare Workstation Pro. Each virtual machine is equipped with 2 processor cores and 4 GB of RAM. These nodes are interconnected using the

Imunes tool [29], a network emulator/simulator capable of replicating realistic network topologies. We employed Imunes to simulate network delays and node failures in our experiments.

Figure 24 depicts the network topology used for the experiments, illustrating the assumed network delays of 10, 20, and 50 ms introduced on the links between routers. These delays mimic the links between the far edge (e.g., LAN), near edge (e.g., a telco-operated Multi-access Edge Computing host) and the cloud. These numbers are roughly based on our round-trip time measurements in (i) a WLAN environment, (ii) over a 5G mobile network connection, and (iii) towards a data center of a global cloud provider in the same continent.

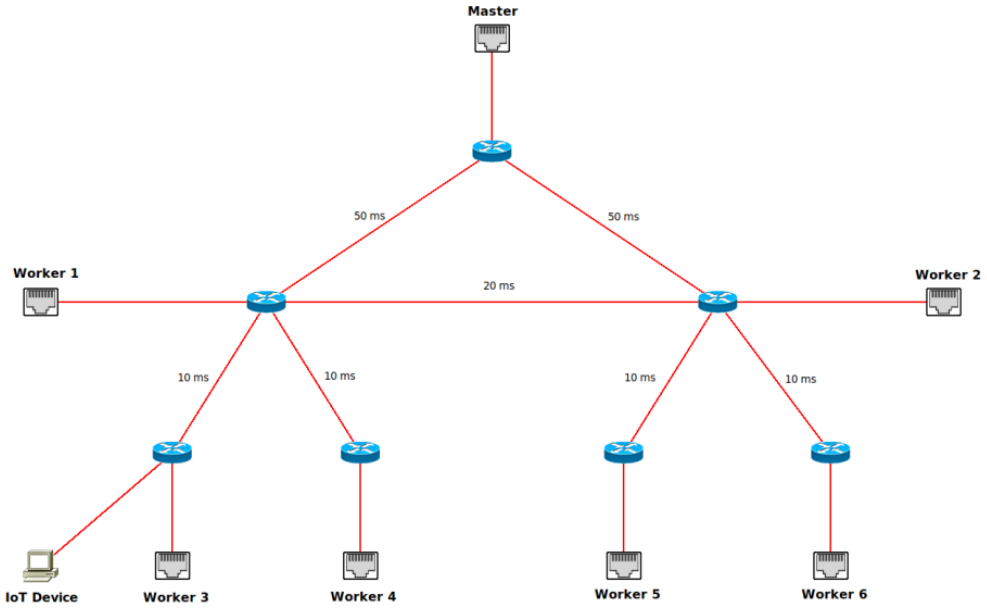


Figure 24: Network topology of the evaluation in the Imunes emulator.

The K3s master node is positioned in the cloud, while nodes *Worker 1* and *Worker 2* are placed at the near edge, closer to the cloud. The remaining four worker nodes and the IoT device are deployed at the far edge layer of the ECC topology. The IoT device is simulated as an Imunes computer and is connected to the same local network as node *Worker 3*.

4.3.2 Experimental Evaluation Scenarios

For the purpose of this evaluation, we generated ≈ 4000 HTTP POST requests from the IoT device to the proxy running on node *Worker 3*. Each request contains a sensor reading for air humidity, temperature, or CO2 concentration, and thus the HTTP request generator acts as a virtual IoT sensor. All IoT device requests are sent to the service responsible for data validation and processing (Data Processing Service, DPS). We defined a single QoS requirement for DPS: its response time must be below **80 milliseconds**. This service notifies the IoT device with a status code about the (un)successful processing of each request, and forwards the processed data to the next service in the data pipeline for advanced analytics, such as an inference service within an AI pipeline for accident predictions. The DPS service instances were deployed within Kubernetes pods on the K3s cluster.

Two scenarios were devised to assess the performance of the previously described configurations:

- **Static scenario**: This scenario evaluates the ability of each configuration to meet the specified QoS requirement under stable conditions.
- **Dynamic scenario:** This scenario evaluates the adaptability of the configurations in handling dynamic changes of the network and service environment. We simulated these changes by increasing network link delays and processing latency, and introducing instance failures within the cluster.

We define DPS to consist of seven instances $S = \{s_1, \dots, s_7\}$ where s_i defines that a service instance is running on node *Worker i*, except instance s_7 which is deployed on the Master node. Since scenarios assume that the data is sent from one IoT device and there is no load coming from other parts of the system, the QEdgeProxy uses a simplified load balancing function which assigns the same weights to all instances in the QoS pool. The initial QoS pool is populated with instances chosen based on the QoS approximations derived from network latency. This pool is continuously updated and maintained using recent latency measurements piggybacked on client requests. Two scenarios were devised to assess the performance of the previously described configurations: static and dynamic scenario.

In the static scenario, seven DPS instances were deployed, one per each node. All requests were sent to the proxy running on node *Worker 3* and during the whole experiment, there were no changes in the network nor in instances deployed in the K3s cluster.

In the dynamic scenario, the state of the pods representing DPS instances or the state of the network is changed after every 750 requests. These changes were designed to simulate real-world events that can impact the QoS of IoT services. The sequence of changes implemented in the dynamic scenario is as follows:

- **T0:** At the start of the simulation, the total of 6 DPS instances are deployed on all nodes except node *Worker 3*.
- **T1:** Instance s_3 is deployed on node *Worker 3*, increasing the total number of DPS instances to seven.
- **T2:** Instance s_3 becomes overloaded, causing its processing delay to increase by 100 ms.
- **T3:** Instance s_3 fails and is removed from the cluster, reducing the number of instances to 6.
- **T4:** Network latency towards instance 1 (on the link between *Worker 1* and its router) is increased by 100 ms.

An overview of the dynamic scenario, i.e., the running state of instances (up/down) at five different points in time is shown in Figure 25. The purpose of this dynamic scenario is to assess the adaptiveness of the three configurations in handling various events in the ECC that can impact QoS: (i) instance joining and leaving the network; (ii) changes in processing latency when a node becomes overloaded; and (iii) changes in network conditions, such as increased latency due to network instability.

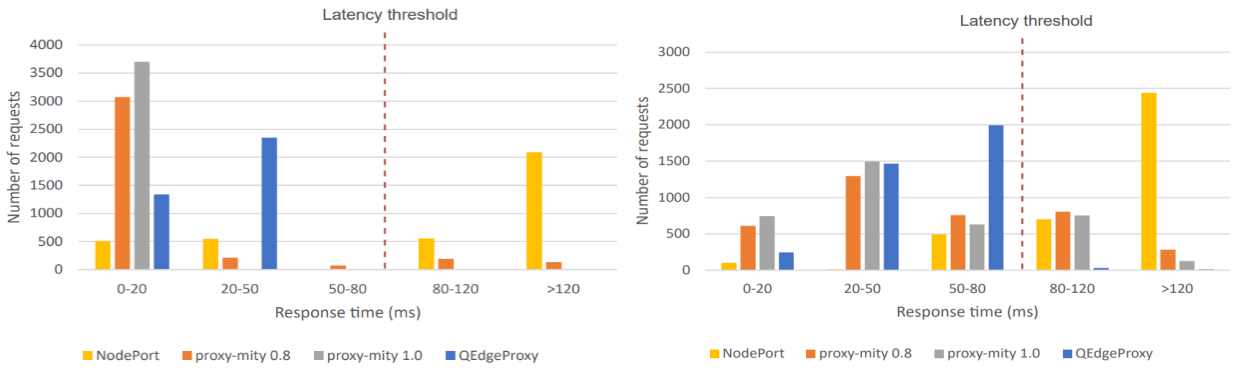
Timestamp	s_1	s_2	s_3	s_4	s_5	s_6	s_7
T0	✓	✓	x	✓	✓	✓	✓
T1	✓	✓	✓	✓	✓	✓	✓
T2	✓	✓	✓↑	✓	✓	✓	✓
T3	✓	✓	x	✓	✓	✓	✓
T4	✓↑	✓	x	✓	✓	✓	✓

↑: increased processing time; ↑: increased network latency

Figure 25: System changes in the dynamic scenario.

4.3.3 Results

The first evaluation parameter is the service response time, as the latency threshold was set as the only QoS requirement for DPS. Figure 26 shows the distribution of response times for the generated 4000 requests as observed for each of the configurations. The graphs categorize response times into five ranges, where the first three ranges (0-20, 20-50 and 50-80 ms) include the requests meeting the QoS requirement of a maximum tolerable response time of 80 ms. The last two ranges represent responses exceeding the acceptable latency limit.



a) Static scenario. *b) Dynamic scenario*

Figure 26: Response time distribution.

In the static scenario (Figure 26a), we can see that the Kubernetes built-in solution NodePort shows by far the worst performance: The largest number of requests handled by NodePort are outside the acceptable QoS range, primarily due to its round-robin load distribution mechanism. This results in a large number of requests being served by instances located far away from the IoT device in terms of network distance, leading to unacceptably high latency.

In contrast, both our QEdgeProxy and proxy-mity 1.0 consistently maintain low response times which meet the QoS threshold, while proxy-mity 0.8 occasionally fails to meet the required QoS limit. This occasional deviation can be attributed to proxy-mity's trade-off between proximity-based routing and load balancing. To ensure balanced load distribution, proxy-mity may occasionally route requests to further-off instances.

Comparing the QEdgeProxy to proxy-mity 1.0, we observe that all requests from proxy-mity 1.0 are directed to the node with the lowest possible latency (range 0-20 ms), while our solution prioritizes meeting the QoS requirements while also performing load balancing. This results in some requests experiencing higher latency, but still within the acceptable range. This can be better observed in Figure 27, where the average response time of proxy-mity 1.0 is significantly lower compared to the QEdgeProxy. However, the number of requests meeting the QoS target, which is the most important parameter for the service client, is nearly identical, favoring proxy-mity by a mere 0.11 %. The QEdgeProxy thus achieves nearly the same QoS guarantees as the purely latency-centric proxy-mity 1.0, while simultaneously performing the load-balancing of requests which reduces the risk of overloading low-latency nodes.

In the dynamic scenario (Figure 26b), NodePort again exhibits the poorest performance due to its static load distribution strategy across all available service instances. Compared to the static scenario, both proxy-mity configurations exhibit a higher proportion of requests exceeding the latency threshold of 80 ms. This is attributed to proxy-mity's inability to detect instance performance variations, such as the increase in processing latency for instance s_3 after timestamp T2: proxy-mity continues to route a majority of requests to instance s_3, resulting in increased latency. Our QEdgeProxy, on the other hand, demonstrates resilience to dynamic changes, consistently maintaining a high proportion of requests within the acceptable threshold. This is evident in Figure 27, where the QEdgeProxy consistently outperforms the other solutions in terms of requests meeting the QoS requirements.

Configuration	**Average response time**		**Successful request rate (QoS)**	
	Static	*Dynamic*	*Static*	*Dynamic*
NodePort	120,11 ms	120,11 ms	28,70%	28,70%
proxy-mity 0.8	20,29 ms	61,05 ms	91,01%	70,78%
proxy-mity 1.0	**6,02 ms**	51.40 ms	**99,97%**	76,27%
QEdgeProxy	26,94 ms	**44,04 ms**	99,86%	**98,86%**

Figure 27: Response time and QoS success comparison.

To evaluate the effectiveness of load balancing capabilities of each configuration, we compare their request distribution across service instances in the static scenario. This comparison is limited to the static setting since the dynamic scenario introduces fluctuating conditions that may distort the load balancing assessment. Instead, we evaluate load balancing in the context of the dynamic scenario in the next subsection which investigates adaptiveness.

Figure 28 shows the distribution of processing instances that served the requests. The instances are sorted based on their network distance from the source node (*Worker 3*) to the node running the instance. NodePort evenly distributes requests across all available instances, adhering to the standard Kubernetes round-robin behavior. Proxy-mity 1.0, on the other hand, sends all requests to the closest instance (s_3), reflecting its prioritization of proximity. Proxy-mity 0.8 exhibits a mixed strategy, with a majority of requests being routed to instance s_3, while occasionally routing requests to other instances, with an almost equal distribution among the remaining instances. In contrast, QEdgeProxy maintains an even load distribution among all three instances within its QoS pool, ensuring continuous adherence to the desired QoS for IoT clients.

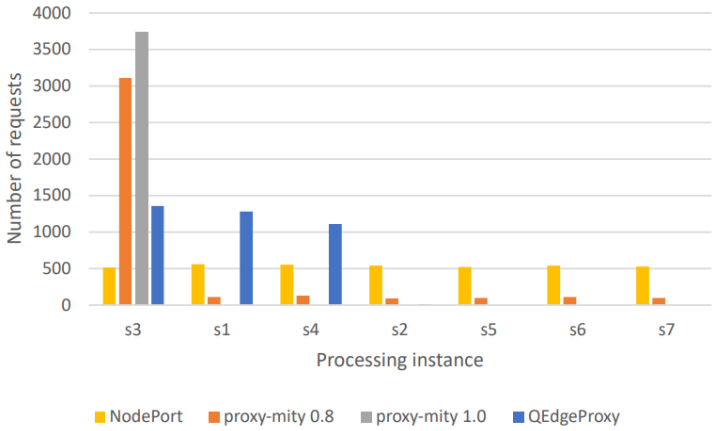
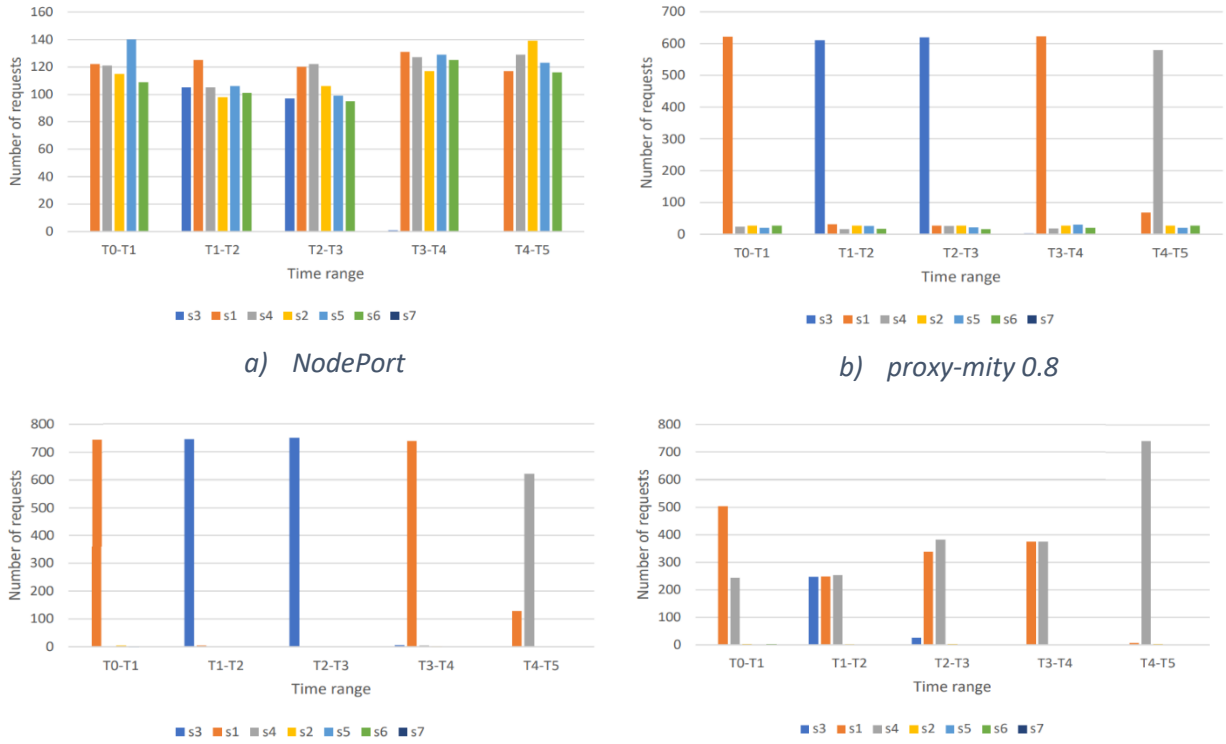


Figure 28: Processing instance distribution: static scenario.

We finally evaluate the ability of the candidate routing solutions to adapt to changes in the ECC, such as node or instance failures, changes in processing load, or changes in the underlying network. We evaluate this through the dynamic scenario.

The graphs in Figure 29 show how the distribution of requests to processing instances changed over time in the dynamic scenario. The time range is set between the timestamps defined in Figure 25.

We can observe in Figure 29a that NodePort, as expected, performs an even distribution of requests to all available service instances for all time ranges. Distribution graphs of both configurations of proxy-mity, Figure 29b and Figure 29c, show that proxy-mity always favors routing to the closest node based on network latency. Therefore, it did not detect the increase in processing latency between timestamps T1 and T2, and continued to send most of the requests to instance s_3 which resulted in high response times and lowered the number of successful requests in terms of meeting the QoS threshold (Figure 27). On the other hand, our QEdgeProxy (see Figure 29d) successfully detected this change in processing latency (by measuring the actual latency) and acted promptly by sending requests only to the remaining instances in the QoS pool, i.e., s_1 and s_4. Also, it can be observed that in the time range T4-T5, when there was an increase in the network latency towards instance s_1, proxy-mity still sent some of its requests to that instance due to the fact that it periodically refreshed the network latency information, while our QEdgeProxy detected the increase in latency with the first requests that failed to meet the QoS requirements. It is important to emphasize that the time to adapt depends on the refresh frequency set in the proxy-mity configuration and the rate of the incoming requests, which indicates that this offset would be lower or non-existent when the request rate is higher than the refresh frequency.



a) NodePort

b) proxy-mity 0.8

c) proxy-mity 1.0

d) QEdgeProxy

Figure 29: Processing instance distribution per time range: dynamic scenario.

Finally, we captured multiple snapshots of the QEdgeProxy's resource consumption under high request load (≈ 1000 requests per second), revealing an average memory consumption of **10 MB**. Considering the K3s agent's memory footprint which ranges from 40 MB, as documented in an idle state on the Raspberry Pi 4 in [15], to 80 MB that we measured in our experiment, we assert that the observed increase in usage remains within an acceptable range. A higher resource consumption compared to proxy-mity which utilizes around 3 MB of memory is expected, as our QEdgeProxy includes the features of maintaining the information on available service pods, QoS pools, and past QoS measurements.

5 Conclusion

This deliverable presents the implementation and evaluation of a data-driven orchestration middleware for energy-efficient IoT supporting ML workflows. The middleware implementation can be divided into two parts: one to support the learning pipeline and another to support the inference pipeline.

FL is an ML technique that allows the model to be trained on a federation of participating devices without the need to centrally store the data which provides a good solution for distributed and resource constrained IoT environment. Hence, to support the learning pipeline, we propose the architecture and mechanisms for adaptive orchestration of FL workflows. The role of the orchestration is to obtain the optimal FL configuration, deploy FL components, and reconfigure the environment if an unexpected event occurs during runtime. The orchestrator was implemented in Golang on top of the Kubernetes container orchestration tool, and the FL services (global aggregator, local aggregator, and client) were implemented in Python as an extension of the Flower framework and packed in a Docker container. The evaluation of the orchestration by training a CNN over a CIFAR-10 dataset shows that the orchestration is stable both on a simulated and real infrastructure. It also demonstrates the ability of the orchestration to quickly react to node state changes by reconfiguring the environment.

To support routing of requests to services in the inference pipeline, we propose the *QEdgeProxy*, an adaptive and QoS-aware load balancing solution specifically tailored for routing client requests to appropriate IoT service instances within the ECC, ensuring uninterrupted data delivery in a dynamically evolving environment. The QEdgeProxy effectively addresses the challenge of maintaining consistent QoS by creating and maintaining the “QoS pool”, a set of service instances that can meet the QoS requirements, and executing load balancing among them. We present a detailed QEdgeProxy implementation within Kubernetes which subscribes to the events in the Kubernetes API to obtain the most recent system changes and to update the QoS pools. The evaluation of QEdgeProxy in a realistic K3s cluster demonstrated its ability to simultaneously perform load balancing and maintain QoS within the acceptable range, outperforming both the built-in Kubernetes load balancing mechanism and a state-of-the-art solution.

For the future work, we plan to extend the implementation of the orchestrator by supporting scheduling and deployment of inference service across the ECC based on the placement of clients that access them. Also, we plan to extend the orchestrator’s scheduler by taking into account the energy consumption of both learning and inference services running on the edge nodes. Current implementation of the adaptive orchestration of FL workflows will be extended with the support for different events that occur in the environment, such as increased node resource consumption or network changes, and with improved reconfiguration decisions based on the reconfiguration cost.

Once we have the full support for an inference pipeline and when inference services are deployed across ECC, we will be able to integrate the QEdgeProxy to route the data to the optimal service. Since both the FL orchestrator and the proxy are designed to run in a Kubernetes environment, we plan to combine them together and allow the routing of inference requests from IoT devices to inference services based on different QoS requirements, such as request latency or accuracy of inference models.

6 Acronyms

AI	Artificial Intelligence
AIoT	Artificial Intelligence of Things
CNN	Convolution Neural Network
D&C	Dissemination and Communication
ECC	Edge-to-Cloud Continuum
FL	Federated Learning
GA	Global Aggregator
IoT	Internet of Things
KLD	Kullback-Leibler Divergence
LA	Local Aggregator
ML	Machine Learning
QoS	Quality of Service

7 List of Figures

8 List of Tables

9 References

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” 2023.

[2] Li, X. Wang, R. Zeng, P. K. Donta, I. Murturi, M. Huang, and S. Dustdar, “Federated domain generalization: A survey,” arXiv preprint arXiv:2306.01334, 2023.

[3] Kairouz et al., Advances and Open Problems in Federated Learning, 2021.

[4] Jianyu Wang, Qinghua Liu, Hao Liang, Gauri Joshi, and H Vincent Poor, “Tackling the objective inconsistency problem in heterogeneous federated optimization”, Advances in Neural Information Processing Systems, 33, 2020.

[5] L. Liu, J. Zhang, S. Song, and K. B. Letaief, “Client-edge-cloud hierarchical federated learning,” in ICC 2020 - 2020 IEEE International Conference on Communications (ICC), 2020, pp. 1–6.

[6] H. Zhu, J. Xu, S. Liu, and Y. Jin, “Federated learning on non-iid data: A survey,” Neurocomput., vol. 465, no. C, p. 371–390, Nov 2021.

[7] Y. Deng, F. Lyu, J. Ren, Y. Zhang, Y. Zhou, Y. Zhang, and Y. Yang, “Share: Shaping data distribution at edge for communication-efficient hierarchical federated learning,” in 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), 2021, pp. 24–34.

[8] Alaa Awad Abdellatif, Naram Mhaisen, Amr Mohamed, Aiman Erbad, Mohsen Guizani, Zaher Dawy, Wassim Nasreddine, “Communication-efficient hierarchical federated learning for IoT heterogeneous systems with imbalanced data”, Future Generation Computer Systems, Volume 128, 2022,

[9] Z. Wang, H. Xu, J. Liu, H. Huang, C. Qiao and Y. Zhao, "Resource-Efficient Federated Learning with Hierarchical Aggregation in Edge Computing," IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, Vancouver, BC, Canada, 2021

[10] Flower Labs GmbH, “Flower A Friendly Federated Learning Framework”, https://flower.ai/

[11]Flower Labs GmbH, "Build a strategy from scratch" https://flower.ai/docs/framework/tutorial-series-build-a-strategy-from-scratch-pytorch.html
[12]FMR LLC, "Spock", https://fidelity.github.io/spock/
[13]"Weights & Biases", https://wandb.ai/
[14]University of Toronto, "CIFAR-10 and CIFAR-100", https://www.cs.toronto.edu/~kriz/cifar.html
[15]I. Čilić, P. Krivić, I. Podnar Žarko, and M. Kušek, "Performance evaluation of container orchestration tools in edge computing environments," Sensors, vol. 23, no. 8, 2023
[16]"K3s - lightweight kubernetes," https://docs.k3s.io/.
[17]L. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "Netmarks: Network metrics-aware Kubernetes scheduler powered by service mesh," in Proc. IEEE INFOCOM, 2021.
[18]J. Santos, T. Wauters, B. Volckaert, and F. D. Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in Proc. IEEE NetSoft, 2019.
[19]K. Tocze, A. J. Fahs, G. Pierre, and S. Nadjm-Tehrani, "Violinn: ´Proximity-aware edge placementwith dynamic and elastic resource provisioning," ACM Trans. Internet Things, vol. 4, no. 1, 2023.
[20]M. L. M. Peixoto, T. A. L. Genez, and L. F. Bittencourt, "Hierarchical scheduling mechanisms in multi-level fog computing," IEEE Trans. Serv. Comput., vol. 15, no. 5, 2022.
[21]J. C. Guevara, R. da Silva Torres, L. F. Bittencourt, and N. L. S. da Fonseca, "Qos-aware task scheduling based on reinforcement learning for the cloud-fog continuum," in Proc. IEEE GLOBECOM, 2022.
[22]L. Angelelli, A. A. D. Silva, Y. Georgiou, M. Mercier, G. Mounie,´ and D. Trystram, "Towards a multi-objective scheduling policy for serverless-based edge-cloud continuum," in Proc. IEEE/ACM CCGrid, 2023.
[23]P. Krivić, M. Kušek, I. Čavrak, and P. Skočir, "Dynamic scheduling of contextually categorised internet of things services in fog computing environment," Sensors, vol. 22, 01 2022.
[24]T. W. Pusztai, S. Nastic, A. Morichetta, V. Casamayor-Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, "Polaris scheduler: SLOand topology-aware microservices scheduling at the edge," in Proc. 15th IEEE/ACM UCC, 2022.
[25]A. Kapsalis, P. Kasnesis, I. S. Venieris, D. I. Kaklamani, and C. Z. Patrikakis, "A cooperative fog approach for effective workload balancing,"IEEE Cloud Computing, vol. 4, no. 2, 2017.
[26]A. J. Fahs and G. Pierre, "Proximity-aware traffic routing in distributed fog computing platforms," in Proc. IEEE/ACM CCGrid, 2019.
[27]Q.-M. Nguyen, L.-A. Phan, and T. Kim, "Load-balancing of kubernetesbased edge computing infrastructure using resource adaptive proxy," Sensors, vol. 22, no. 8, 2022.
[28]"Istio service mesh," https://istio.io/latest/about/service-mesh/
[29]"IMUNES: integrated multiprotocol network emulator/simulator," http://imunes.net/.

