AIoTwin

Twinning action for spreading excellence in Artificial Intelligence of Things

DELIVERABLE D1.4

DEMONSTRATOR OF NEW MECHANISMS FOR EDGE ORCHESTRATION, ENERGY-EFFICIENCY AND ML WORKFLOWS

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.	**D1.4**
Title	Demonstrator of New Mechanisms for Edge Orchestration, Energy-Efficiency and ML Workflows
Lead Editor	Mislav Has
Type	Report
Dissemination Level	PU/SEN
Work Package	WP1
Due Date	31.10.2024

AUTHOR(S)

Name	Partner	e-mail
Mario Kušek	UNIZG-FER	mario.kusek@fer.hr
Ivana Podnar Žarko	UNIZG-FER	ivana.podnar@fer.hr
Ivan Čilić	UNIZG-FER	ivan.cilic@fer.hr
Mislav Has	UNIZG-FER	mislav.has@fer.hr
Ana Petra Jukić	UNIZG-FER	ana-petra.jukic@fer.hr
Katarina Vuknić	UNIZG-FER	katarina.vuknic@fer.hr
Fehmi Ben Abdesslem	RISE	fehmi.ben.abdesslem@ri.se

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
0.01	20-10-2024	Mario Kušek	Initial structure
0.02	02-10-2024	Ana Petra Jukić, Katarina Vuknić	Energy consumption of different devices during model training in a federated learning environment
0.03	04-11-2024	Ivan Kralj	Comprehensive analysis of the benefits and drawbacks of Spatio-Temporal Graph Neural Networks trained in semi-decentralized environment
0.04	06-11-2024	Mislav Has	Power consumption experiment of WebAssembly runtime on Raspberry PI devices
0.05	21-11-2024	Ivan Kralj	Assessment of the practical applicability of semi-decentralized ST-GNNs for real-world deployment scenarios and future directions

0.06	4-12-2024	Ivan Čilić	Framework for adaptive orchestration of HFL pipelines and Reconfiguration validation algorithm (RVA)
0.07	6-12-2024	Ivan Čilić	Describing experimental evaluation of the framework and RVA
0.08	9-12-2024	Katarina Vuknić Ana Petra Jukić	Results of performance measurements of FL with edge devices
0.09	9-12-2024	Mislav Has	Introduction, Executive Summary and Conclusion
0.10	9-12-2024	Mario Kušek	Internal review
0.11	10-12-2024	Mislav Has	Version sent to Fehmi Ben Abdesslem for internal review
0.12	12-12-2024	Fehmi Ben Abdesslem	Reviewing and writing comments and suggestions
0.13	18-12-2024	Ivan Kralj	Applied reviewer comments on spatio-temporal GNNs
0.14	18-12-2024	Ivan Čilić	Applied reviewer comments on adaptive orchestration of HFL pipelines
1.0	03-01-2025	Ivana Podnar Žarko	Final version for submission.

Table of Contents

Executive Summary

This deliverable presents a demonstrator of advanced mechanisms for adaptive orchestration, energy efficiency, and machine learning (ML) workflows within federated learning (FL) environments. The demonstrator consists of the following three parts: adaptive orchestration of hierarchical FL pipelines, evaluation of spatio-temporal graph neural networks in FL, and energy consumption and performance evaluation in FL setups and WebAssembly runtimes on edge devices. The first part focuses on optimizing FL workflows through the development of the Reconfiguration Validation Algorithm (RVA) and a framework for adaptive orchestration, showing improvements in efficiency and flexibility. The second part explores spatio-temporal graph neural networks and gossip learning for decentralized FL, with experimental results comparing training setups and evaluating cloudlet performance. The third part includes energy consumption measurement on edge devices (Jetson AGX Orin, Jetson Nano, and Raspberry Pi 4) in FL setups and performance evaluation of WebAssembly runtimes (Wasmtime, Wasmer, WasmEdge) for computational tasks like matrix multiplication and ML inference, offering insights into energy optimization and runtime efficiency. Together, these parts demonstrate the integration of orchestration, energy efficiency, and performance optimization for ML workflows in edge computing, providing valuable insights for the AIoTwin orchestration middleware.

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 focusing 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, D1.3 focused on the practical implementation of the outlined requirements and proposed architecture, with key decisions regarding federated learning frameworks and orchestration tools informed by the demonstrator's conclusions.
This deliverable, **D1.4**, advances the AIoTwin framework with novel mechanisms for efficiency and flexibility of FL workflows by focusing on three key areas: the adaptive orchestration of hierarchical federated learning workflows, including reconfiguration validation and experimental evaluations, the exploration of spatio-temporal graph neural networks in decentralised setups, and the measurement of energy consumption in FL and WebAssembly runtimes on edge devices.

1.2 Deliverable Outline

This document is organized as follows. Section 2 focuses on the adaptive orchestration of hierarchical FL pipelines, including reconfiguration validation, the orchestration framework, and experimental evaluations. Section 3 explores spatio-temporal graph neural networks, detailing traditional, server-free, and gossip learning setups, along with their experimental evaluation. Section 4 examines energy consumption measurements, with a focus on FL on edge devices and WebAssembly runtime implementations. Section 5 concludes the deliverable, while Section 6 provides a list of acronyms. Finally, the lists of figures, tables, and references are presented in Sections 7, 8, and 9, respectively.

2 Adaptive Orchestration of Hierarchical Federated Learning Pipelines

Contrary to traditional FL where there is a single global aggregator (GA) typically residing in the cloud, hierarchical federated learning (HFL) introduces *local aggregators* (LAs) at the edge which are used as intermediary aggregators between the clients and global aggregator residing typically in the cloud.

In a typical HFL setup, an ML engineer defines an **HFL task** that runs within an HFL pipeline in the Computing Continuum (CC[1]). The HFL task is defined with the following inputs: (i) an initial ML model, (ii) training parameters, and (iii) the **orchestration objective**. The initial ML model serves as the starting point from which all clients begin their training during the initialization phase of the pipeline. The training parameters include batch size, learning rate, etc. An orchestration objective is defined on a case-by-case basis, for example to either optimize the model's performance within a predefined cost budget, e.g., a specified communication cost budget, or to minimize the cost while achieving a specific target, such as a target level of model accuracy or loss. Other HFL performance parameters can be used when defining the orchestration objectives and such diversity should be supported by an HFL orchestrator.

The dynamic CC environment where HFL operates suggests that changes will occur during the execution of an HFL pipeline, such as nodes joining and leaving the pipeline, node resource fluctuations, or network disruptions. Such events can impact the performance of a running HFL pipeline. For example, a client becoming unavailable can introduce degradation of the ML model performance, or a network change can increase the communication cost of the pipeline, and this can faster deplete a potential cost budget defined for a HFL task. Responding to such events requires an **adaptive orchestration mechanism** that maintains the HFL pipeline deployed with the best HFL configuration for the given state of the CC. Carrying out the appropriate **reconfigurations** is however non-trivial, particularly given that applying a new configuration at runtime is not guaranteed to provide a better ML model performance, as we show next.

2.1 Reconfiguration validation

2.1.1 Motivation

Let us consider a scenario where an HFL pipeline is deployed with two clusters, each of them consisting of five nodes. At round R^{reconf} a new node joins the CC which holds data available for training. In Scenario A the node that joins holds a high number of dataset samples and introduces lower communication cost per global round, while in Scenario B the joining node has a small dataset and introduces higher communication cost per global round.

Figure 1 and Figure 2 show the effect of reconfiguration when the objective of an HFL task is to provide the best model in terms of accuracy within the available communication cost budget. The upper graphs show how accuracy changes over global rounds, while the lower graphs depict cost changes over global rounds. The performance of the new configuration is shown with a yellow line. Two scenarios

[1] Computing Continuum is a multi-layered infrastructure that spans from IoT and edge devices to more powerful intermediate nodes, such as edge servers, and finally to the cloud infrastructure. The continuum enables seamless data processing and service delivery across different layers, from the edge, where the data is generated, to the cloud, where computational power and large-scale data management are available.

demonstrate examples when a reconfiguration introduces either a model improvement (Scenario A) or a model degradation (Scenario B). On one hand in Scenario A, the final model accuracy was higher with the new configuration when communication cost budget is reached, meaning that the reconfiguration has indeed introduced an improvement of the model performance. On the other hand, in Scenario B, the original configuration would reach a higher accuracy compared to the new configuration within the same communication cost budget, which means that the applied reconfiguration introduced performance degradation. Thus, it would be better not to include the new node with a small dataset into the pipeline in Scenario B.

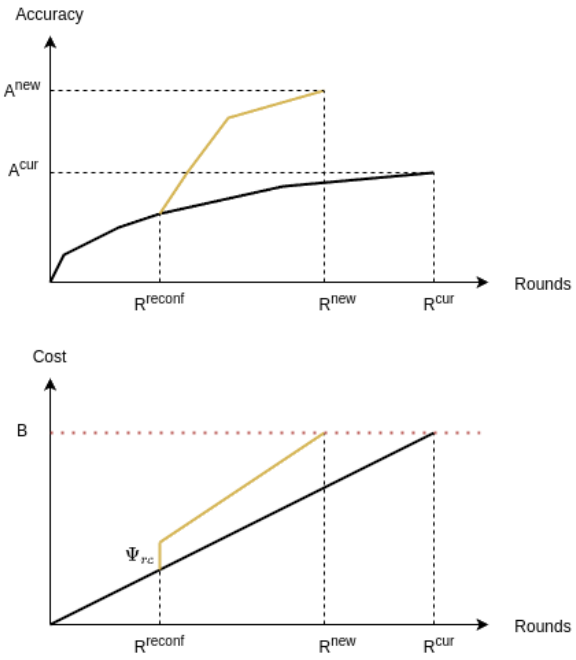


Figure 1. Reconfiguration effect: scenario A

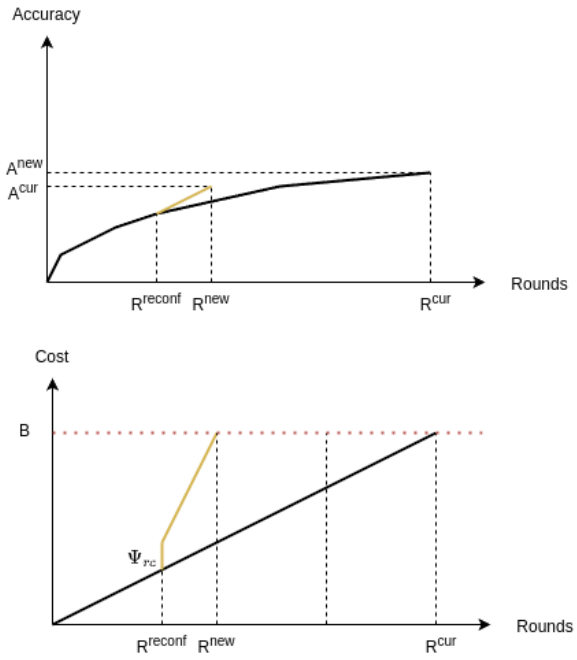


Figure 2. Reconfiguration effect: scenario B

In both scenarios, the new configuration improves the model performance; however, in Scenario B the new configuration introduced a slower performance improvement, since the new node held a smaller dataset, and the cost per-round was much higher, resulting in the cost budget being quickly exceeded.

2.1.2 Reconfiguration validation algorithm (RVA)

A HFL pipeline reconfiguration can introduce performance degradation in terms of model performance or increased cost. For this reason, we present a reactive reconfiguration validation approach named *Reconfiguration Validation Algorithm* (RVA) to identify such situations. The RVA checks the reconfiguration status after a specific number of global rounds post-reconfiguration, denoted as the validation window *W*, and reverts the pipeline to the original configuration if needed. Figure 3 shows an example of the sequential flow of events during reconfiguration validation and depicts how model accuracy changes during HFL global rounds. An event occurs at moment R^{reconf} and the reconfiguration is performed at the end of the current global round. After that, the orchestrator waits for *W* rounds until validation round R^{val} when it performs reconfiguration validation to check whether the pipeline should

be reverted to the original configuration. In this example, the validation decision is based on the observation that the latest reconfiguration introduces model performance degradation, and the original configuration is reverted.

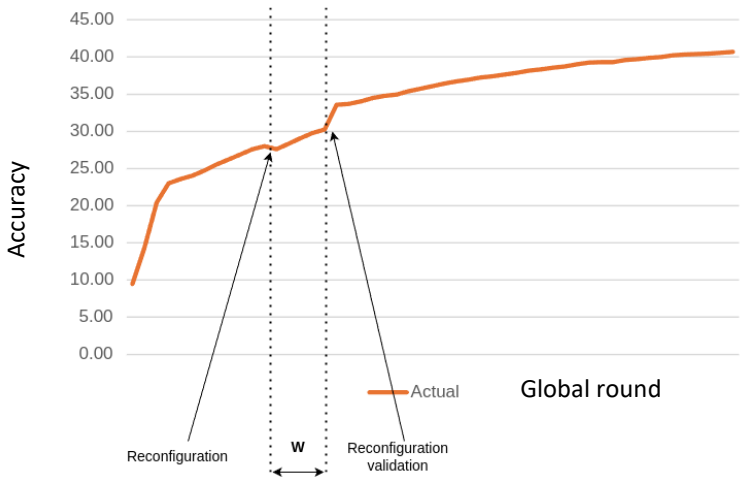


Figure 3. RVA: execution flow

The decision on whether a configuration introduces model performance improvement or degradation is made based on the prediction of the future behaviour of the original and new configuration. This prediction is made through regression based on the actual model performance values. An example of regression functions used for performance prediction is shown in Figure 4.

We can see that the prediction of the original configuration is made based on the performance values before reconfiguration, and the prediction of the new configuration is made based on the values obtained during the validation window. After calculating the regression functions for both configurations, the orchestrator predicts the performance value of **the round when the budget is exceeded**. If the predicted value with the original configuration is higher than the one for the new configuration, the original configuration is reverted.

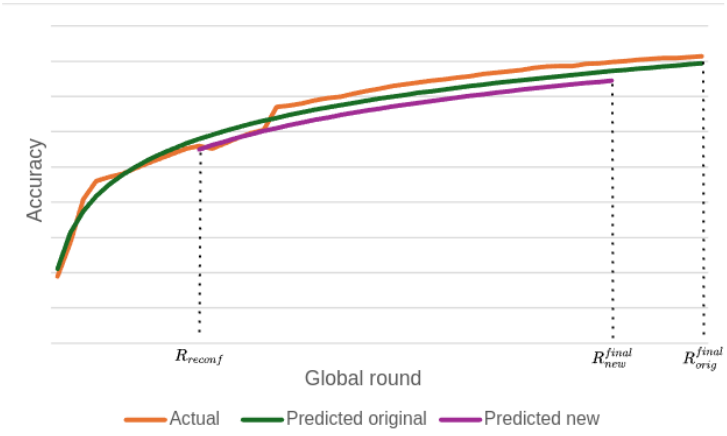


Figure 4. RVA: performance prediction

More details on the algorithm implementation and calculation of reconfiguration cost can be found in the following paper (currently under review, submitted to a conference call for papers) [1].

The adaptive orchestration of HFL pipelines with the developed AIoTwin orchestration middleware which implements the RVA was demonstrated to the audience of the 2nd AIoTwin Summer School held in Dubrovnik in September 2024.

2.2 Framework for adaptive orchestration of hierarchical federated learning pipelines

The "architecture for adaptive orchestration of FL workflows" proposed in report D1.3 (Figure 1) is implemented as an extension of Kubernetes, an industry-standard container orchestration tool, with the focus on orchestrating HFL pipelines. Hence, the implementation, named *framework for adaptive orchestration of HFL pipelines,* supports HFL components running in containers. Figure 5 shows how the framework is built on top of the Kubernetes tool.

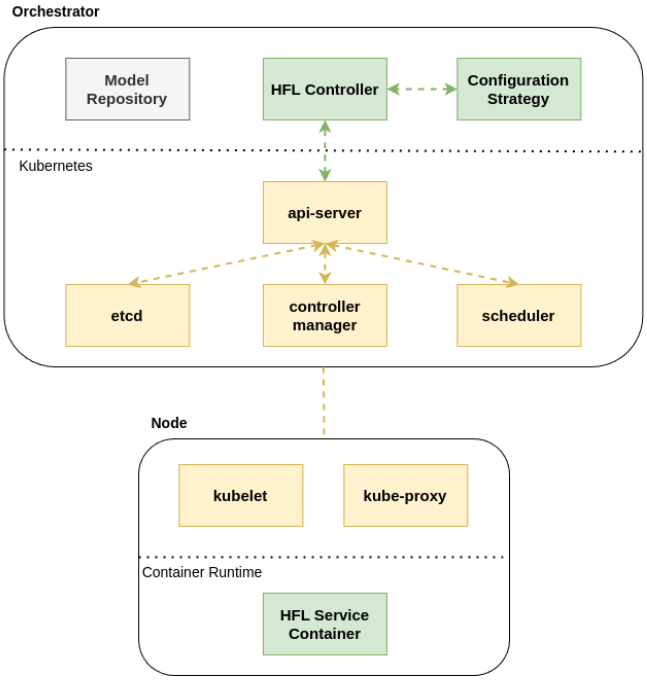


Figure 5. Framework for adaptive orchestration of HFL pipelines

The implementation details about the framework are reported in deliverable D1.3 and are now extended with an implementation of reconfiguration validation. The framework currently supports reconfigurations triggered by events when a node state changes (nodes joining or leaving an HFL pipeline) and orchestration objectives defined in terms of a limited communication cost budget. The reconfiguration validation algorithm is executed after *W* rounds, a parameter that can be configured by a user. For the prediction of model performance, we currently use logarithmic regression, as it has shown to be the most suitable for predicting the behaviour of model accuracy, which grows quickly in the beginning and then slows down.

2.3 Experimental evaluation

To evaluate the orchestration framework and RVA, we conducted experiments on a K3s cluster, a lightweight distribution of Kubernetes [1]. We deployed a cluster of nine (from 9 up to 14) nodes[2]: one controller and eight (from 8 up to 13) worker nodes. All the nodes were virtual machines running *Ubuntu Server 22.04* with 2 CPU cores and 4 GB of RAM.

All experiments consisted of training an untrained CNN model on a CIFAR-10 dataset. The CNN architecture used has two convolutional layers (6 and 16 channels, both having a kernel size of 5, a stride of 1 and padding 0) each followed by a ReLU activation function and a 2x2 max pooling. After flattening the output of the last layer, the model includes two fully connected layers with 120 and 84 units respectively, both using ReLU activation functions. Finally, the model ends with an output layer of 10 units, for each class in CIFAR-10.

The dataset on each client was specified for each experiment separately to show the difference in performance when having different data distributions. All the experiments used communication as the cost parameter.

The step-by-step instructions on how to run these experiments is given in the following repository: https://github.com/AIoTwin/fl-orchestrator/tree/icmlcn.

2.3.1 Evaluation target: framework performance

The first evaluation scenario evaluates the performance of the implemented framework. Therefore, it evaluates the orchestrator deployment on the actual infrastructure, i.e., each node is hosting one FL service, either a client, LA, or GA. The network cost between the nodes and the data distribution on the nodes are shown in Figure 6. Data distributions are depicted per each client to denote a class label and number of samples of this class.

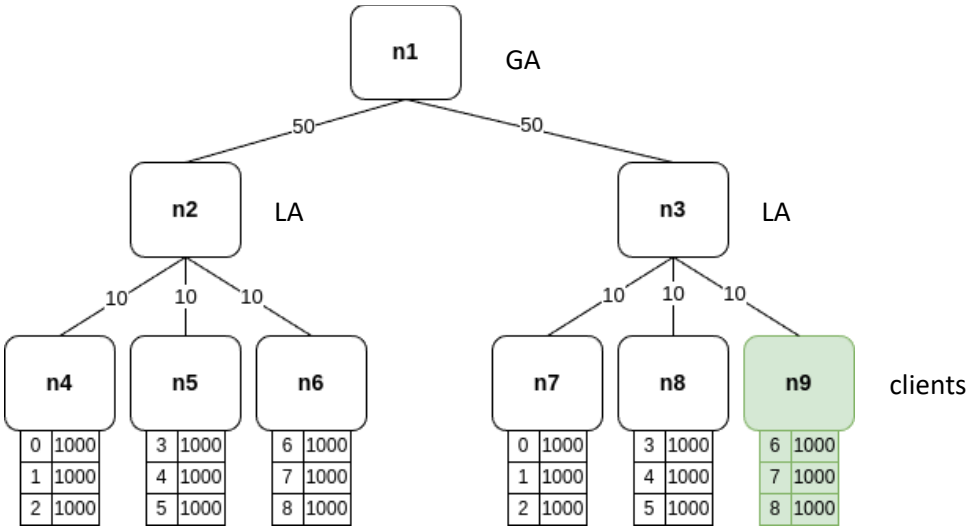


Figure 6. Deployed topology to evaluate the framework

[2] The total number of applied nodes depends on the experimental setup.

We can see that the clusters are balanced in terms of data distribution, i.e., both clusters cover the same classes from 0 to 8, and contain the same number of samples per class. Node n_9 is coloured in green because it was the node that has left the running pipeline and then rejoined the pipeline during the experiment. Table 1 shows the evaluation parameters for the first experiment. The cost was defined with a total budget and the clustering of nodes was created so that clusters minimize the communication cost.

Table 1. Performance evaluation: configuration parameters

Cost Configuration Type	Total Budget
Communication Budget	50 000
Configuration Strategy	*minCommCost*
Local Epochs	2
Local Rounds	2

In the beginning, all the nodes from topology in Figure 6 were available and the FL pipeline was started. After **10** global rounds node n_9 left the system, which resulted in node removal from the pipeline. Then, after round **20** node n_9 rejoined the system which resulted in reconfiguring back to the original configuration. Note that for this evaluation RVA was disabled as this experiment is evaluating the framework performance. Figure 7 and Figure 8 show how accuracy and communication cost were changing over time.

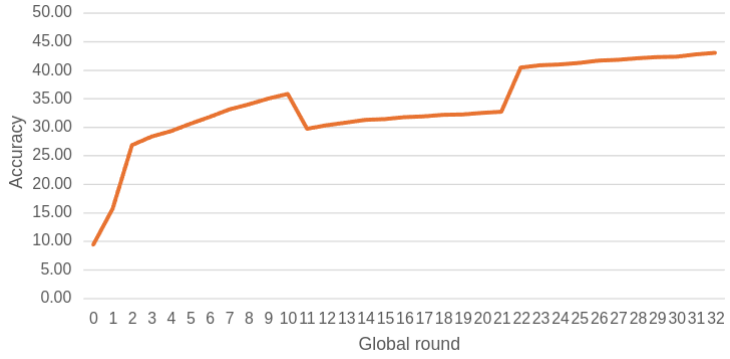


Figure 7. Performance evaluation: accuracy

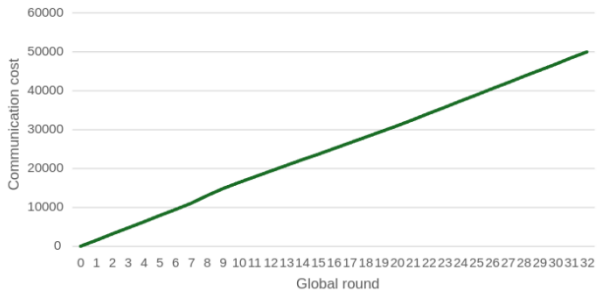


Figure 8. Performance evaluation: total cost

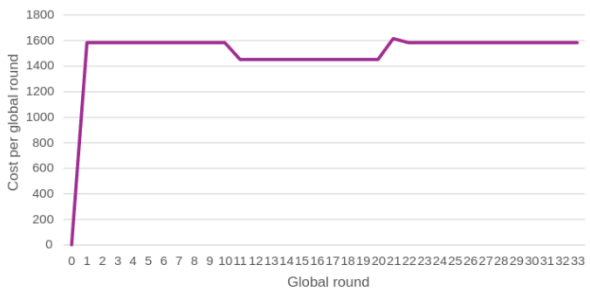


Figure 9. Performance evaluation: cost per round

We can observe that after round 10 when node n_9 was removed, there was a significant drop in performance, which was then improved after round 20 when the node rejoined. Regarding the

communication cost, we can see that the training stops when the budget was reached (Figure 8). Also, in Figure 9 we can observe that the cost per global round dropped when n_9 was removed from the pipeline which is expected as there is one less model update in the pipeline. Additionally, there is a slight increase in the cost between rounds 20 and 21 because now the reconfiguration change cost occurs due to the fact that n_9 needs to download the latest model from its parent aggregator n_2. Note that there is no reconfiguration change cost after round 10 because removing a node doesn't generate communication cost.

Finally, Table 2 shows the results of the performance evaluation of our framework. The results show that the FL orchestrator introduces minimal computing overhead to the system because it only runs the configuration model and subscribes to the events in the Kubernetes API. Also, the results show that the orchestrator needs significantly more time to detect a new node joining a pipeline than to detect node removal. This is a limitation of K3s as it needs some time to detect the "Ready" state of a node. However, it quickly detects node removal, which is more important as it can act quickly to run the pipeline with the optimal configuration without the removed node.

Table 2. Performance evaluation: benchmark results

FL Orchestrator CPU Usage	≈ 0.15 cores
FL Orchestrator Memory Usage	≈ 15 MB
Node Joining Reaction Time	≈ 15 s
Node Leaving Reaction Time	≈ 500 ms

2.3.2 Evaluation target: RVA

Next, we evaluate the RVA with two types of data distribution on client nodes: IID and non-IID. In the IID setup, all clients had an equal data distribution, containing all classes, and the equal number of samples per class. In the non-IID setup, each client had two classes in its dataset and the number of samples per class was equal. Table 3 shows configuration parameters used in the evaluation. The reconfiguration trigger event was a new node (or multiple nodes) joining the system, the reconfiguration round R^{reconf} was set to 10, and the validation window W was set to 5.

Table 3. RVA evaluation: configuration parameters

Cost Configuration Type	Total Budget
Communication Budget	100 000
Configuration Strategy	*minCommCost*
Local Epochs	2
Local Rounds	2
Reconfiguration trigger event	New Node(s)
R^{reconf}	10
W	5

2.3.2.1 RVA exp. #1: IID setup with performance degradation

Figure 10 shows the environmental setup of the experiment showing performance degradation in the IID environment. At R^{reconf} two nodes n_{10} and n_{11} join the system and bring in small datasets (the changes

are coloured in green). The nodes are attached to clusters of n_2 and n_3, respectively. The small dataset refers to the dataset with all 10 classes and 100 samples per class.

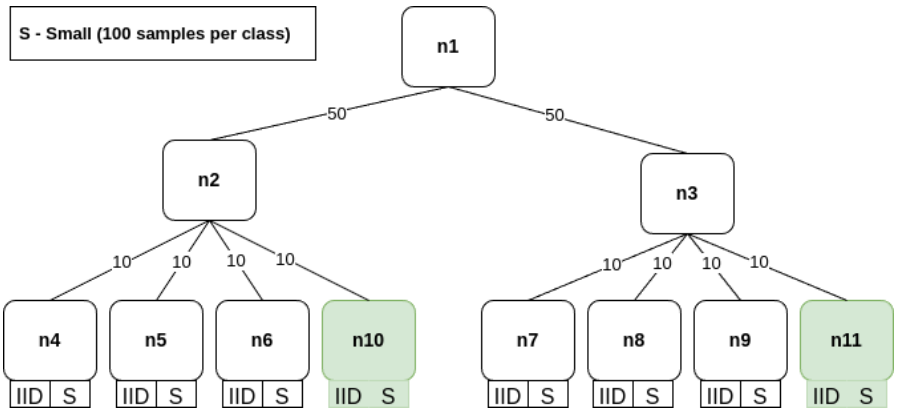


Figure 10. RVA exp. #1: environmental setup

The evaluation results show the trend of accuracy and total cost over global rounds for two scenarios: with RVA and without using RVA. We can see in Figure 11 that there is no significant improvement in performance during the reconfiguration validation window (between R^{reconf} and R^{val}) and that the communication budget is exceeded sooner if we keep the new configuration (RVA-disabled in Figure 12). Therefore, the RVA decided to revert to the original configuration at R^{val} which proved to be a good decision as the scenario with RVA enabled reached a higher accuracy within the allocated communication budget than the scenario with RVA disabled.

R_{rva}^{final} and $R_{rva-disabled}^{final}$ show the final rounds of both scenarios when the available communication budget has been used.

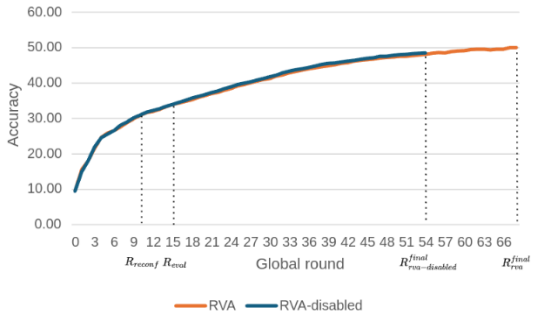


Figure 11. RVA exp. #1: accuracy

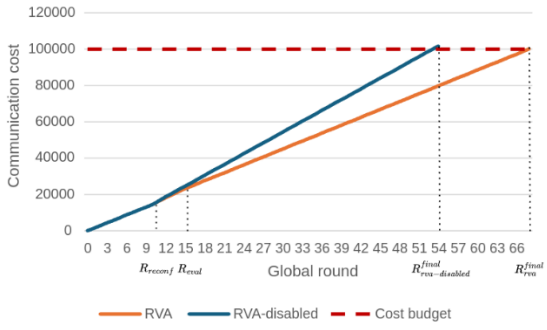


Figure 12. RVA exp.#1: total cost

2.3.2.2 RVA exp. #2: IID setup with performance improvement

Figure 13 shows the environmental setup of the experiment showing performance improvement in the IID environment. Like in RVA exp. #1, at R_{reconf} nodes n_{10} and n_{11} join the pipeline, but in this experiment,

they hold large datasets. A large dataset refers to the dataset with all 10 classes and 1000 samples per class.

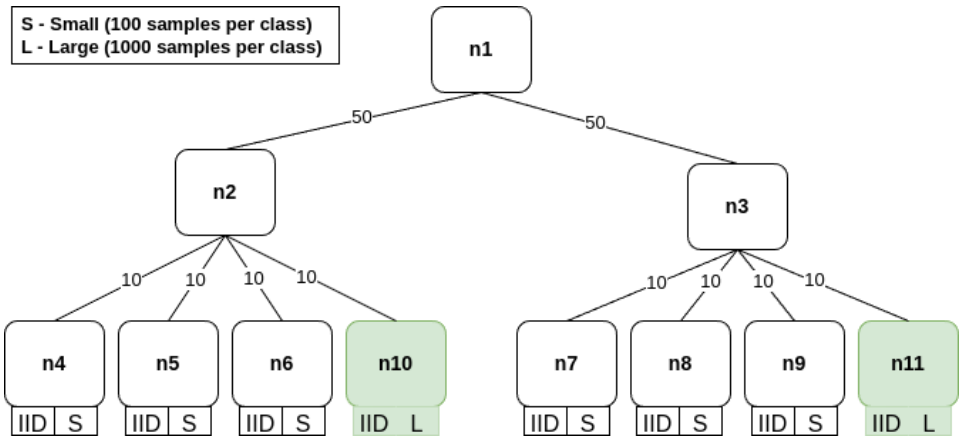


Figure 13. RVA exp. #2: environmental setup

The results in Figure 14 and Figure 15 show the improvement in model performance after the reconfiguration round R_{reconf}. This improvement was detected during the validation window, between R_{reconf} and R_{val}. In Figure 14 we can see the predicted performance with the original configuration made with regression: its final round happens after the final round with the new configuration because the original configuration had a lower communication cost per round, as depicted in Figure 15. However, it was predicted that the final accuracy value when the budget exceeds would be higher with the new configuration, so the new configuration was kept. Also, in Figure 15 we can see how the cost per round increases after the reconfiguration as we have two additional node updates in each round. Also, we can see the cost of reconfiguration change as a spike which occurs during the round R_{reconf}.

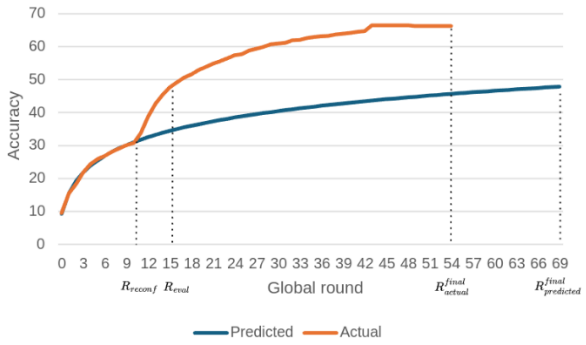


Figure 14. RVA exp. #2: accuracy

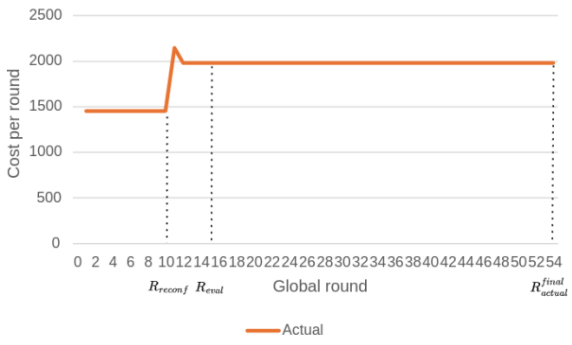


Figure 15. RVA exp. #2: cost per round

2.3.2.3 RVA exp. #3: Non-IID setup with performance degradation

Figure 16 shows the environmental setup of the experiment showing performance degradation in the non-IID environment. At R_{reconf} a node n_{14} joins the cluster of node n_2 with a dataset representing classes *0* and *1*, which are already covered within the cluster.

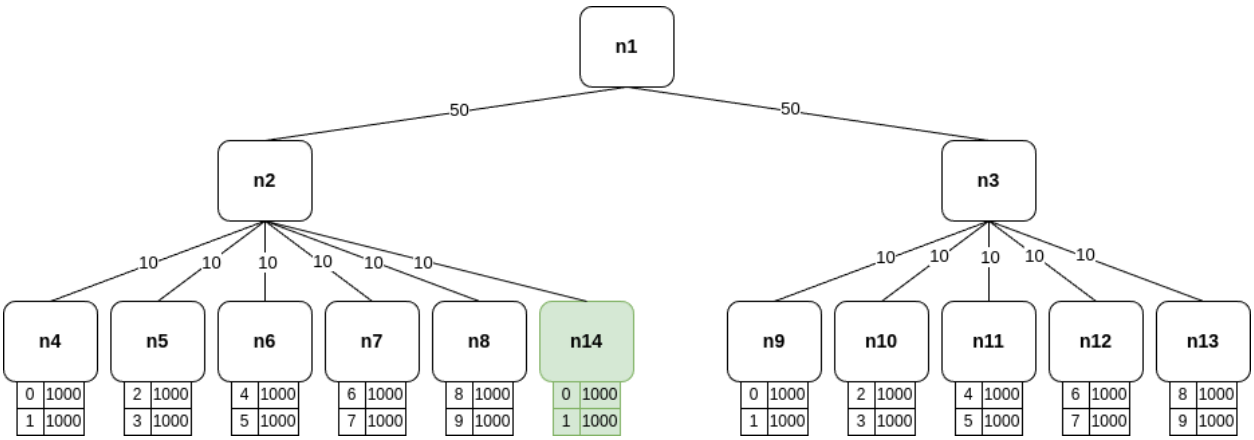


Figure 16. RVA exp. #3: environmental setup

The evaluation results show the trend of accuracy and total cost over global rounds for two scenarios: with RVA and without RVA. In Figure 17 we can see that there is even a drop in performance after the reconfiguration when n_{14} joins the cluster, which is a result of clusters being imbalanced. Therefore, in R_{val} RVA decides to revert the pipeline to the original configuration without node n_{14}. We can see that the RVA made the right decision, because it outperforms the scenario without using RVA which would keep the new configuration. In Figure 18 we can see how the total cost is changing over global rounds. We can see that in scenario with RVA enabled the budget is reached later than in scenario without RVA because the RVA reverted to the original configuration at round R_{reconf} and the original configuration is less costly per round since it does include model updates from node n_{14}.

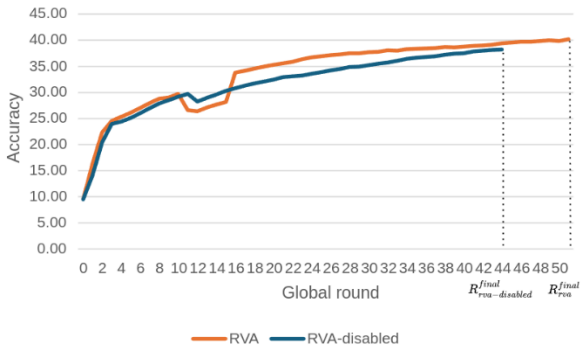


Figure 17. RVA exp. #3: accuracy

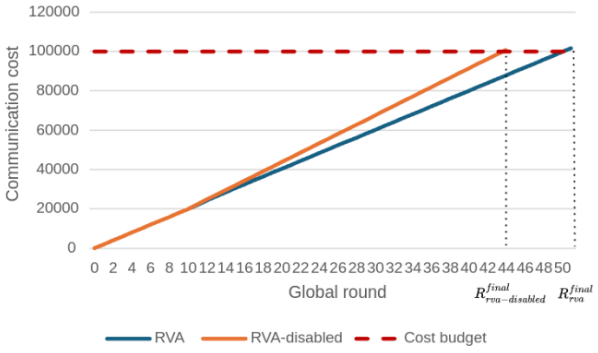


Figure 18. RVA exp. #3: total cost

2.3.2.4 RVA exp. #4: Non-IID setup with performance improvement

Figure 19 shows the environmental setup of the experiment showing performance improvement in the non-IID environment. At R_{reconf} node n_{13} joins the cluster of node n_3 with a dataset representing classes *8* and *9*, which are not present in the cluster. This makes both clusters balanced with equal data distributions.

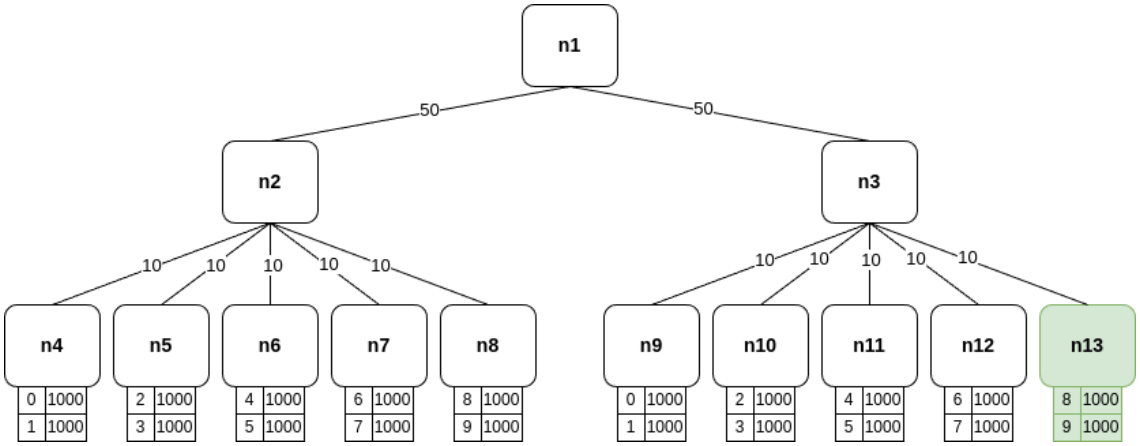


Figure 19. RVA exp. #4: environmental setup.

The results in Figure 20 and Figure 21 show the improvement in model performance after the reconfiguration round R_{reconf}. This improvement is detected at R_{val} and RVA decides to keep the new configuration. In Figure 20 we can see how RVA made a prediction that keeping the new configuration would produce better performance than reverting to the original even though the original configuration would reach the budget at a later global round. In Figure 21 we can see the growth in cost per round after reconfiguration and also a spike at R_{reconf} which is due to the added cost of reconfiguration change.

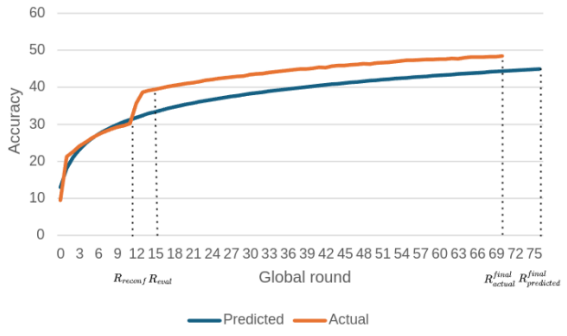


Figure 20. RVA exp. #4: accuracy.

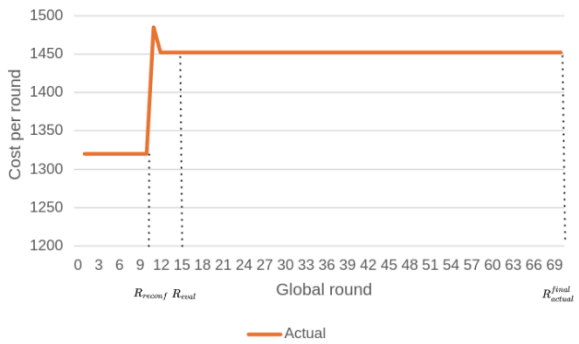


Figure 21. RVA exp. #4: cost per round.

3 Spatio-Temporal Graph Neural Networks

Graph Neural Networks (GNNs) [4] are special types of neural networks capable of working with a graph data structure, where nodes represent entities or individual data points in the graph, and edges represent the relationship or interactions between pairs of nodes. This representation allows GNN to capture spatial changes by considering the interconnectedness of nodes within the network. However, GNNs cannot capture both spatial and temporal dependencies, hence Spatio-Temporal Graph Neural Networks (ST-GNNs) have been developed. The ability to capture both spatial and temporal dependencies enables the model to simultaneously account for spatial correlations between nodes and temporal correlations over time, making it particularly suited for spatio-temporal forecasting tasks like traffic prediction.

Multiple ST-GNNs have been developed **Error! Reference source not found.** , such as Graph Recurrent N etwork (GRN) [6], Spatio-Temporal Graph Convolutional Network (ST-GCN) [7], Graph Attention LSTM Network [8], and many more [5]. Each technique addresses specific challenges or offers unique advantages over others. The model used in this project is the Spatio-Temporal Graph Convolutional Network (ST-GCN) due to their superior ability to handle traffic prediction tasks in the IoT context compared to other techniques by achieving the best performance with statistical significance in all evaluation metrics (MAE, MAPE, and RMSE) [7].

However, previous works have mainly focused on ST-GNNs trained in centralized environments, where data from all sensors is continuously collected. However, the task of collecting geo-distributed data, performing deep learning training and inference, and sending back commands, is challenging to perform reliably in real-time in a centralized system. As sensor networks expand—either by covering larger, more distant regions or by increasing their density to provide more granular sensing in urban areas— scaling a central control system to handle the increased volume of data in real-time with low-latency becomes increasingly difficult. Furthermore, any issues in the central control system can easily result in a complete interruption of the service across the whole sensor network, requiring significant investments to maximize reliability. Given the key role played by mobility infrastructure in our societies, a centralized system can also become a significant target for hostile actors and a major cyber-weakness in strategic infrastructure management.

To the best of our knowledge, there has been no prior research specifically focused on decentralized or even semi-decentralized training of ST-GNNs for traffic prediction. One relevant work by L. Giaretta et. al. [9] explores the challenges and solutions associated with fully decentralized training. However, their solution is not tailored to ST-GNNs, but rather to Graph Convolutional Network (GCN), limiting its applicability to tasks that require both spatial and temporal correlations. While there is no prior research work for decentralized training for ST-GNNs, M. Nazzal et. al. [10] developed Heterogeneous Graph Neural Network with LSTM (HetGNN-LSTM) for taxi-demand prediction and tested it in a semi-decentralized environment, where nodes were partitioned into cloudlets, without any central aggregator. However, their solution only enables semi-decentralized inference by deploying a pre-existing model, with no support for semi-decentralized training.

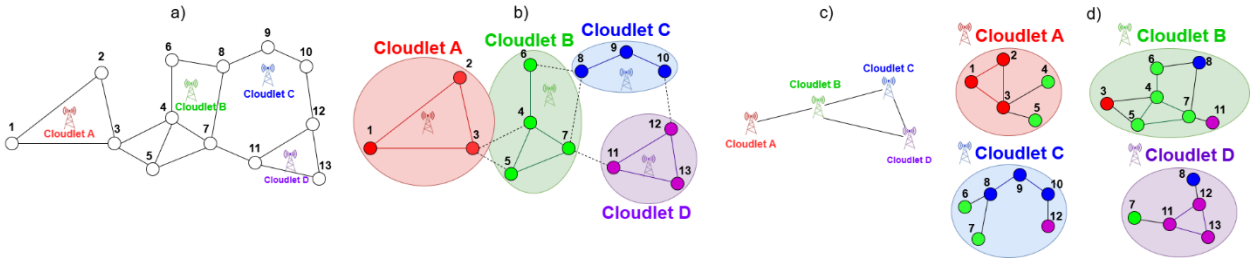


Figure 22. Graph partitioning and communication. a) Geographically distributed sensor network and base stations b) Graph partitioning of the sensors into cloudlets based on geographical proximity. c) Cloudlet-to-cloudlet communication network for exchanging node features and model updates. d) After communicating with neighbouring cloudlets, each cloudlet can construct the ST-GNN subgraph required for training on its local nodes (reported in [11])

Hence, we adopt a semi-decentralized ST-GNN architecture inspired by Nazzal et al. [10], as it's one of the most effective architectures for distributed training. Figure 22 shows key components of such architecture. In Figure 22-a, traffic network is represented as a graph, where nodes correspond to IoT-devices, and edges represent road connections based on physical proximity. In Figure 22-b, the graph is partitioned into subgraphs based on geographical proximity, each managed by a local cloudlet. In Figure 22-c, cloudlets form a communication network for exchanging necessary node features and model updates. After communicating with each other, each cloudlet constructs ST-GNN subgraph required for training on its local nodes, as shown in Figure 22-d. Using this architecture, we evaluate four different training setups — centralized, traditional FL, server-free FL, and Gossip Learning — for ST-GNNs in the context of traffic prediction, with the centralized setup serving as a baseline for comparing the semi-decentralized approaches.

3.1 Traditional Federated Learning

Traditional FL [12] is a collaborative machine learning (ML) approach where multiple clients work together to train a model, coordinated by a central aggregator. Instead of sharing raw data, which remains stored locally on each client device, the clients exchange only model updates with the server. These updates contain just the minimum information needed for the learning task, carefully scoped to limit the exposure of client data. The central aggregator then aggregates these focused updates as soon as they are received, ensuring that the learning process is both efficient and compliant with data minimization principles.

3.2 Server-free Federated Learning

Server-free FL [13] is a decentralized variant of the traditional FL approach. Unlike traditional FL, where a central aggregator is responsible for coordinating the model updates, server-free FL operates without a central entity. Instead, participating devices communicate directly with one another to exchange model parameters and perform local updates. After initializing their models, devices exchange model parameters with their neighbours, aggregate these received models with their own local model, and proceed to train on a subset of data. This iterative process allows for continuous refinement of models across the network, as the updated models are exchanged again between neighbouring devices.

3.3 Gossip Learning

Gossip Learning [14] is a decentralized protocol for training ML models in distributed settings without the need for a central coordinator. It is designed to be highly scalable and robust, allowing model updates to

be propagated efficiently across a network of devices or nodes. Each device stores two models in its memory (as a FIFO buffer), typically representing the most recent models received from neighbouring devices. At each iteration of the gossip protocol, a device averages the weights of these two models to create an aggregated model. The device then performs one local training step using its own data, refining the aggregated model based on its local observations. Once the model is updated, the device forwards it to a randomly chosen device in the entire network. This process is repeated across the network, ensuring that models continuously evolve and improve as they traverse different nodes, collecting knowledge from the data they encounter at each location.

3.4 Experimental Setup

All experiments were run with a fixed number of 40 epochs, utilizing the High-Performance Computing (HPC) system Leonardo Booster, which features 4x NVIDIA A100 SXM6 64GB GPUs, with only 1 being used during training. The software environment used for the experiments included Python 3.10.14, PyTorch 2.3.1, and PyTorch Geometric 2.5.3.

For model architecture, we used 2 spatio-temporal blocks (ST-blocks), with GLU activation function, a learning rate of 0.0001, dropout of 0.5, weight decay of 0.00001, and a batch size of 32. MAE is used as the training loss, Adam was chosen as the optimization algorithm, and learning rate scheduler was applied with the StepLR strategy, where the step size was set to 5, and the decay factor was set to 0.7. The temporal and spatial kernel size was set to 3 for all experiments, with Chebyshev convolution as convolutional layer. All experiments use 60 minutes as the historical time window, i.e., 12 observed data points.

Additionally, sensors were distributed across 7 cloudlets based on proximity and communication range. Each cloudlet can communicate with other cloudlets and IoT-devices only if they are within an 8 km range. This limitation directly impacts the server-free FL approach by restricting the number of cloudlets that can exchange model updates. In contrast, this constraint does not affect Gossip Learning, as model updates are sent to a randomly selected cloudlet across the entire network, regardless of proximity. Figure 23 shows how sensors are assigned to cloudlets and their respective communication ranges for both the METR-LA and PeMS-BAY datasets. This partitioning reflects the semi-decentralized structure used for our distributed training approach.

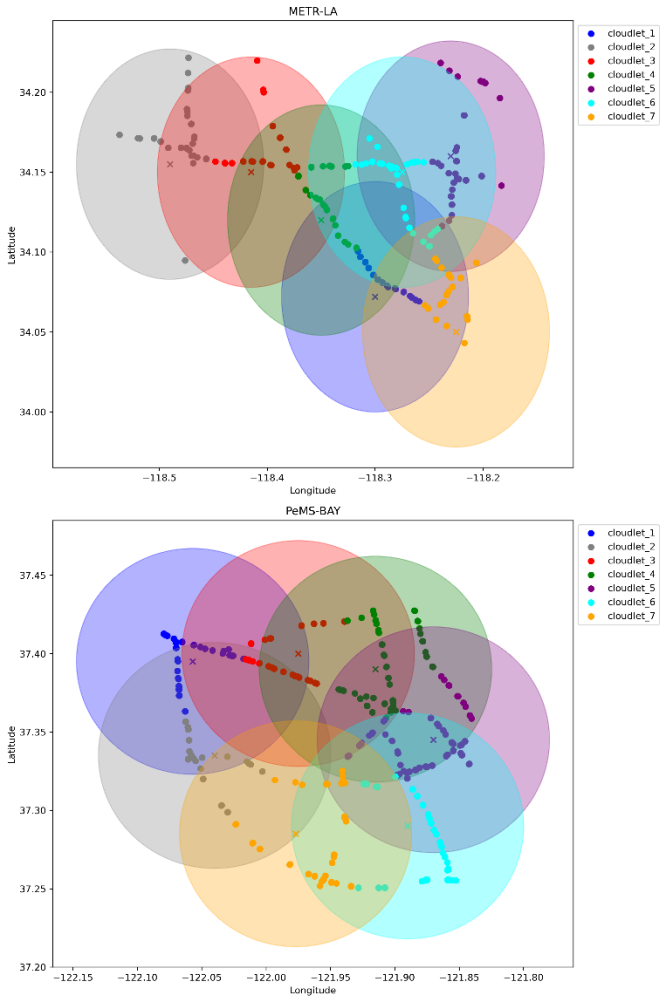


Figure 23. Sensor assignment to cloudlets based on communication range

3.4.1 Datasets

To evaluate the performance of our setups, we utilize two real-world public traffic datasets, PeMS-BAY and METR-LA, which are collected by Caltrans performance measurement system. Both datasets consist of traffic data aggregated in 5-minute intervals, with each sensor collecting 288 data points per day. The details of datasets are listed in Table 4.

Table 4 Details of the METR-LA and PeMS-BAY datasets

Datasets	Nodes	Interval	TimeSteps	Attribute
METR-LA	207	5 min	34,272	Traffic speed
PeMS-BAY	325	5 min	52,166	Traffic speed

The spatial adjacency matrix is constructed from the actual road network based on distance using the formula from ChebNet [15]. Specifically, it encodes the spatial relationships between nodes by assigning weights to edges based on the inverse of the pairwise distances between connected nodes, effectively capturing the connectivity and proximity of locations in the traffic network. Additionally, the data were

pre-processed for efficient model training and evaluation. For example, if we want to utilize the historical data spanning one hour to predict the data the next hour, we pack the sequence data in group of 12 and convert it into an instance. The dataset was then split into training, validation, and test sets in a 70:15:15 ratio. Additionally, to mitigate the impact of varying data magnitudes, we applied standardization to normalize the original data values, reducing the impact of the large difference in value.

3.4.2 Evaluation Metrics

In our experiments, we use three commonly adopted metrics for traffic forecasting to evaluate the performance of the different setups: Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and Weighted Mean Absolute Percentage Error (WMAPE). The formulas are as follows:

$$MAE(x, \hat{x}) = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

$$RMSE(x, \hat{x}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)}$$

$$WMAPE(x, \hat{x}) = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{\sum_{i=1}^{n} |y_i|}$$

Where $x = x_1,\ldots,x_n$ denotes the ground truth values of vehicle speed, and $\hat{x} = \hat{x}_1,\ldots, \hat{x}_n$ represents their predicted values.

We compute these metrics across three forecasting horizons — short-term (15 min), mid-term (30 min) and long-term (60 min) — to capture performance variations over time. For centralized and FL setups, metrics are derived from the main server's aggregated model, while for server-free FL and Gossip Learning, they are computed as a weighted average of individual cloudlet predictions. This ensures a consistent and fair comparison across all approaches.

3.5 Experimental Results

3.5.1 Comparison of training setups

Table 5 presents the results of four training setups, across both datasets for three forecast horizons. The table highlights that the centralized setup consistently outperforms the semi-decentralized methods across all evaluation metrics and forecast horizons. The observed performance gap is extremely small between all 4 setups, especially for MAE and RMSE. For example, in the METR-LA dataset, the difference in MAE between the centralized setup and the best semi-decentralized setup is 0.1 miles per hour for short-term prediction, which is negligible compared to the difference between normal traffic flow and a traffic jam, typically measured in tens of miles per hour. While WMAPE shows a slightly more noticeable gap as the forecast horizon increases, particularly for the METR-LA dataset, the difference between centralized and semi-decentralized methods remains within a narrow margin across all horizons. In other words, this difference is insignificant to disregard the competitive performance of semi-decentralized approaches compared to the centralized one, indicating no practical drawbacks in adopting semi-decentralized methods for traffic prediction use-case.

Table 5. Performance comparison (MAE [mile/h]/RMSE [mile^2/h^2]/WMAPE [%]) of four different setups [11]

Datasets	Setups	15 min			30 min			60 min		
		MAE	RMSE	WMAPE	MAE	RMSE	WMAPE	MAE	RMSE	WMAPE
METR-LA	Centralized	3.78	9.05	7.45	5.14	11.61	10.12	7.35	14.59	14.47
	Traditional FL	3.97	9.01	7.82	5.40	11.41	10.63	7.82	14.48	15.40
	Server-free FL	3.90	8.98	7.78	5.35	11.37	10.67	7.79	14.41	15.55
	Gossip Learning	3.88	9.04	7.74	5.43	11.35	10.85	7.56	14.42	15.10
PeMS-BAY	Centralized	1.48	3.09	2.38	1.97	4.23	3.17	2.59	5.41	4.17
	Traditional FL	1.50	3.12	2.42	2.03	4.29	3.27	2.67	5.51	4.29
	Server-free FL	1.50	3.12	2.42	2.01	4.25	3.23	2.65	5.44	4.28
	Gossip Learning	1.51	3.12	2.43	2.03	4.28	3.26	2.63	5.39	4.24

3.5.2 Cloudlet metric analysis

Figure 24 illustrates the variability in WMAPE across individual cloudlets for all training setups and both datasets. The results are shown for short-term and long-term prediction horizons, providing a comprehensive view of how errors distribute geographically across cloudlets for different prediction horizons. We focus exclusively on WMAPE because all metrics exhibit similar patterns of spread across cloudlets. However, WMAPE is particularly insightful as it highlights the relative error in percentage terms, making it easier to interpret and compare the variability across different cloudlets.

From the figure, we observe consistent spread in WMAPE values across cloudlets, irrespective of the training setup, dataset, or prediction horizon. This consistency suggests that the variability in cloudlet performance is not an artifact of the training setups, but is instead intrinsic to the geographical characteristics of the datasets and sensor partitioning, alongside the values of data itself. Each cloudlet's performance is influenced by the distribution of sensors assigned to it and the corresponding traffic patterns in that geographical region.

Additionally, while global weighted averages reported in Table 5 provide a summary of performance, they fail to capture the significant differences in performance across individual cloudlets. Previous works have primarily focused on global averages, overlooking notable challenge of heterogeneous performance among cloudlets. This oversight can lead to an incomplete understanding of model performance in distributed traffic forecasting systems, resulting in unexpected discrepancies, suboptimal decision-making and potential disruptions in real-world usage compared to testing.

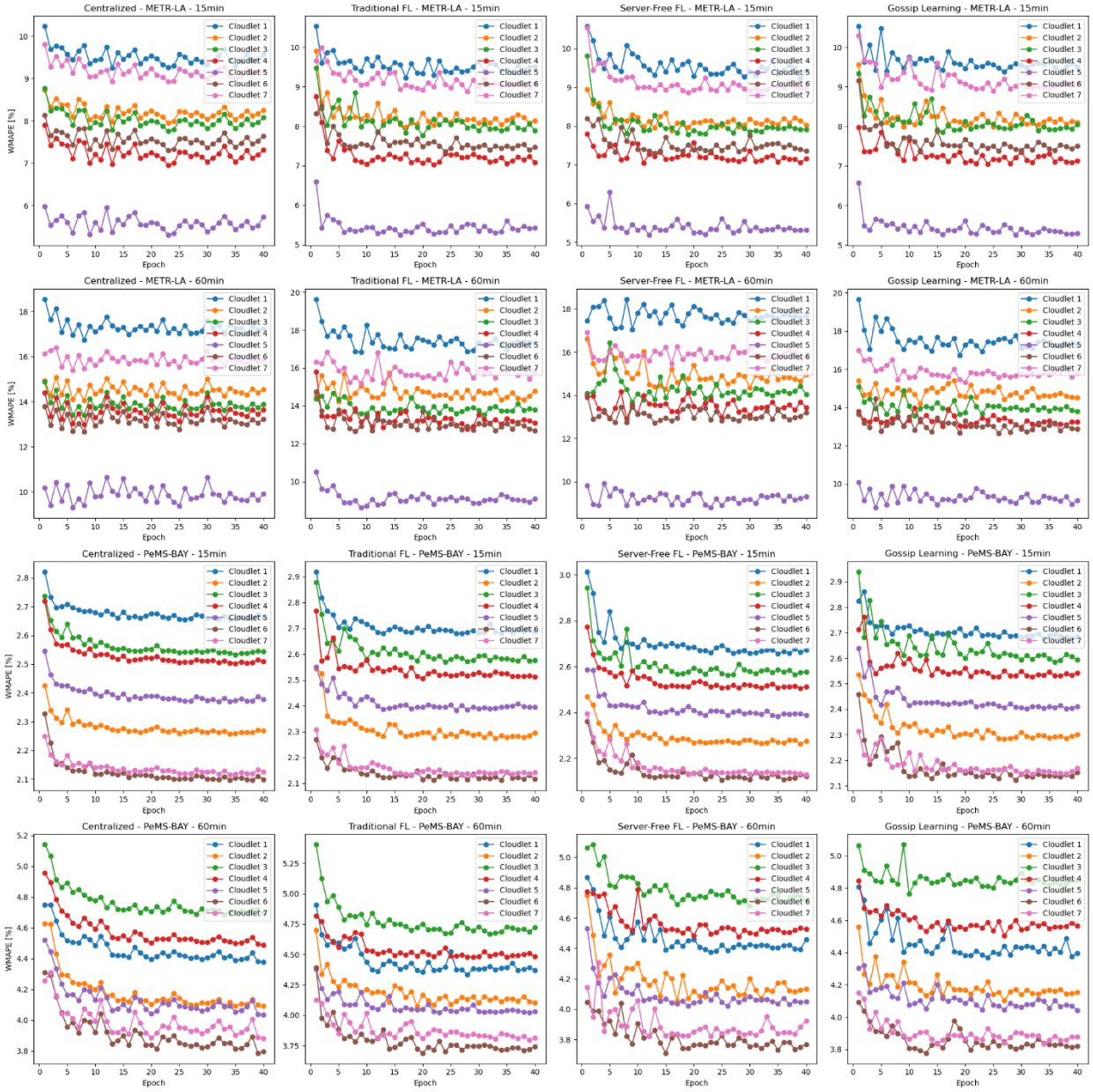


Figure 24. WMAPE for individual cloudlets (reported in [11])

3.5.3 Analysis of semi-decentralized overheads

Semi-decentralized learning naturally introduces several overheads, primarily stemming from the need to communicate and aggregate models, increasing communication and computational costs compared to centralized training. Additionally, due to graph partitioning, node features must be exchanged between cloudlets, further contributing to communication and compute demands. Table 6 breaks down these overheads across three key aspects: model transfer, node feature transfer, and floating-point operations (FLOPs).

Table 6. FLOPs and average model and node feature transfer size per cloudlet [11]

Datasets	Setups	Model [MB]/epoch	Training FLOPs/epoch	Aggregation FLOPs/epoch	Node feature [MB]
METR-LA	Centralized	-	0.24T	-	0.68
	Traditional FL	0.13	1.56T	0.16M	3.69
	Server-free FL	0.23	1.56T	0.25M	3.69
	Gossip Learning	0.13	1.56T	0.42M	3.69
PeMS-BAY	Centralized	-	0.58T	-	1.61
	Traditional FL	0.13	3.89T	0.16M	9.44
	Server-free FL	0.26	3.89T	0.56M	9.44
	Gossip Learning	0.13	3.89T	0.42M	9.44

One of the most significant communication costs in semi-decentralized training is the transfer of node features. Unlike centralized training, where each sensor's features are sent once to a central server, distributed setups require features to be transferred to potentially multiple cloudlets. This increase is due to our distance-based weighted adjacency matrix creating a relatively dense graph, where each sensor node is connected to all other nodes within a predefined radius, and the use of a 2-hop GNN, meaning that each cloudlet not only needs features from its direct neighbours but also from the neighbours of those neighbours, significantly expanding the portion of the graph that each cloudlet must process. Consequently, each cloudlet must store and process a substantial portion of the entire graph, leading to a several-fold increase in communication compared to centralized setups. However, as the network grows larger, the portion of the graph stored in each cloudlet will decrease, as cloudlets farther away will not have connected sensors. Despite the increase, the overall feature transfer per cloudlet remains manageable, typically just a few megabytes.

In contrast to node feature transfers, model transfer per epoch remains relatively small across all distributed setups. This is primarily due to the compact size of the ST-GCN models used in our experiments. While model transfers accumulate over multiple epochs, leading to higher total communication overhead, the per-epoch cost remains in the order of megabytes. As shown in Figure 25, distributed setups, particularly server-free FL, require more epochs to converge than centralized training, compounding the total model transfer cost. Despite this, the overall communication overhead from model transfers remains modest and does not present a significant bottleneck. Even in the worst-case scenario of server-free FL, where all cloudlets potentially communicate with each other, the model transfer overhead remains manageable.

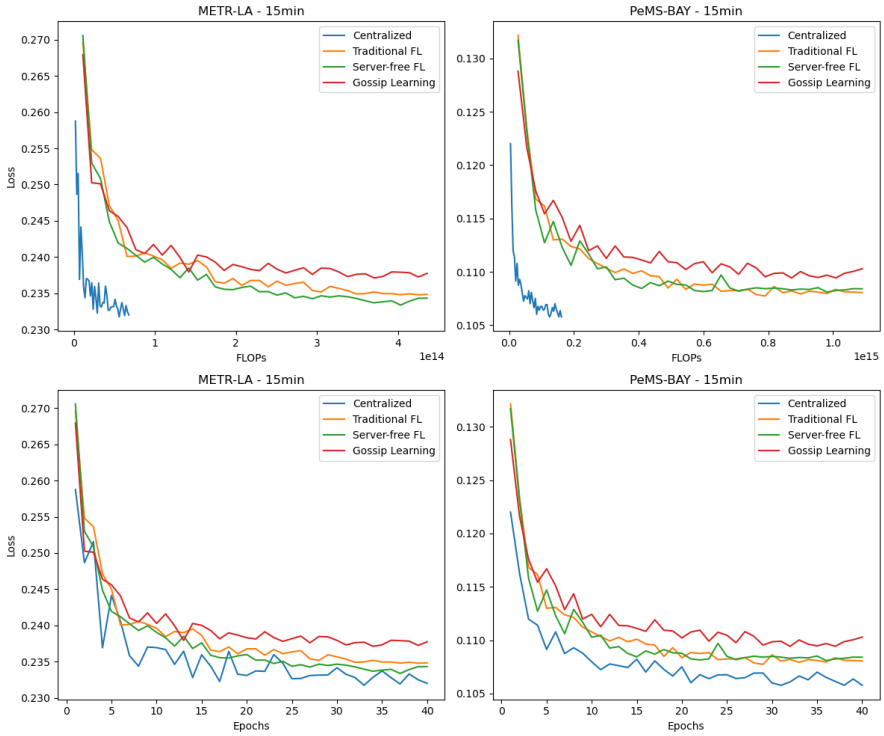


Figure 25. Validation loss over FLOPs and epochs for short-term prediction (reported in [11])

In terms of computation, we see that aggregation costs are many orders of magnitude smaller than training costs, so they do not represent a reason to avoid semi-decentralized training. In contrast, the training costs in distributed setups are several times higher than in centralized training. This mirrors the pattern observed in feature transfer, as the root cause is similar. Each cloudlet must run the GNN on its local subgraph, which often includes duplicated nodes and features due to the overlapping receptive fields of neighbouring cloudlets. This duplication leads to additional computations, as cloudlets must compute partial embeddings for nodes they do not have to facilitate GNN message passing within their local subgraph. Only the cloudlet that owns a node can compute its full k-hop embedding and make accurate predictions. In contrast, a centralized system can efficiently train over the entire graph without such redundancies, avoiding duplicated computations.

Our analysis reveals that the major sources of overhead in semi-decentralized training do not stem from the distributed learning algorithms themselves. If these algorithms were applied to non-spatial models, the overheads would be minimal. Instead, the overheads arise from the highly interconnected spatial nature of traffic data and the need to partition these spatial connections across cloudlets. Overall, the trade-off between the flexibility and scalability of semi-decentralized training and the increased communication and computational overheads highlights the challenges of maintaining spatial dependencies in distributed setups. While the communication and computation costs are higher than centralized training, they remain within acceptable limits, making semi-decentralized approaches a feasible solution for scalable traffic prediction in smart mobility applications.

We report further details about the experimental setup and results presented this section in the following paper (currently under review, submitted to a conference call for papers) [11].

4 Measuring energy consumption

4.1 FL with edge devices

For this demonstrator, we utilize the CIFAR-100 dataset, distributed non-identically and non-independently across clients. To accommodate this setup, we implement a different model compared to the one used in Section 2.3, the Wide ResNet. Regarding the issue described in the previous demonstrator (D1.2) about the utilization of GPU of the Jetson Nano device, we have solved it with the assistance of a Docker image containing all the installations compatible with the JetPack SDK 4.4.1, including PyTorch.

The evaluation of energy consumption of different devices during the AI model training in a FL setup included using a **semi-large model with 8.6 million learnable parameters for image classification** on a dataset with 100 classes, distributed across 8 edge devices. The setup includes **one Jetson AGX Orin, five Jetson Nanos, and two Raspberry Pi 4 devices**. The Jetson AGX Orin utilizes its CPU, the Raspberry Pi devices employ CPUs, and four Jetson Nanos use GPUs while one uses a CPU for training the FL model. The experiment uses a CIFAR-100 dataset and the WideResNet model, implemented in Python with the PyTorch and Flower framework. The Dirichlet distribution is applied to simulate real-world data distribution. Power and energy consumption for each device are measured using the WLAN power socket throughout the learning process.

CIFAR-100 is a widely used computer vision dataset for benchmarking image classification models. It contains 100 classes, each with 500 training images and 100 testing images. The 100 classes are grouped into 20 superclasses containing various aspects of nature, infrastructure, animals, people, vehicles, food and household items. The images have dimensions of 32x32 and are in RGB format. To simulate a realistic FL environment, the employed data distribution across clients was non-IID and created by scattering data from each class in uneven quantities. The Dirichlet distribution is utilized to simulate non-IID data across clients by adjusting class proportions. Its parameter alpha controls the level of skewness: smaller values create more skewed distributions, while larger values result in more uniform (IID) distributions. An alpha of 0.5 is chosen as the parameter for the Dirichlet distribution in this experiment. Figure 26 illustrates the non-IID setup across clients, with each bubble representing the number of images containing a specific class label. The largest bubble corresponds to approximately 400 data points.

Figure 26. Data distribution across clients: non-IIID allocation, bubble size shows the number of images per class for each client

WideResNet (Wide Residual Network) [3] is a modified version of the ResNet that contains wider convolutional layers (number of channels), with reduced depth (number of layers). This architecture allows for faster training time and improved performance with skip connections effectively addressing the vanishing gradient issue. Reduced computational overhead makes WideResNet better suited for training on edge devices compared to more complex state-of-the-art networks. The model architecture is shown in Table 7.

Table 7. WideResNet model architecture

Layer	**Output Shape**	**Details**
Input	[3,32,32]	Input image
Conv2D	[16, 32, 32]	Initial convolution with 16 channels
Basic Block (4)	[16, 32, 32]	4 x Residual Blocks, each consisting of two convolutional layers with BatchNorm after each. Output of the block: skip connection links the block's input directly to the output of the second convolutional layer, with ReLU activation applied after the addition.
Basic Block (5)	[160, 16, 16]	5 x Residual Blocks

Basic Block (5)	[320, 8, 8]	5 x Residual Blocks
Linear layer	[100,]	Fully connected layer for classification into 100 classes.
Total params	-	**8,662,100** total parameters

To compare energy consumption, four devices are connected to a WLAN power socket. These include one Jetson AGX Orin, which uses its CPU to train the model, two Jetson Nanos (one utilizing its CPU and the other its GPU), and a Raspberry Pi 4 utilizing its CPU. The specifications of the devices are provided in Table 8.

Table 8. Hardware specifications of the devices

Device - model	**CPU**	**GPU**	**RAM**
Jetson AGX Orin Developer Kit	12-core Arm Cortex A78AE v8.2	2048-core NVIDIA Ampere architecture GPU with 64 Tensor Cores	64GB
Jetson Nano	Quad-core Arm Cortex-A57	128-core NVIDIA Maxwell architecture GPU	4GB
Raspberry Pi 4 Model B	Quad core Arm Cortex-A72	-	4GB

The training was conducted over 15 rounds, with each client participating in every round, and each round consisting of two epochs. Figure 27 contains plots generated by TensorBoard, showing training loss, training accuracy, test loss and test accuracy. The final test accuracy reached during the experiment was approximately 43%.

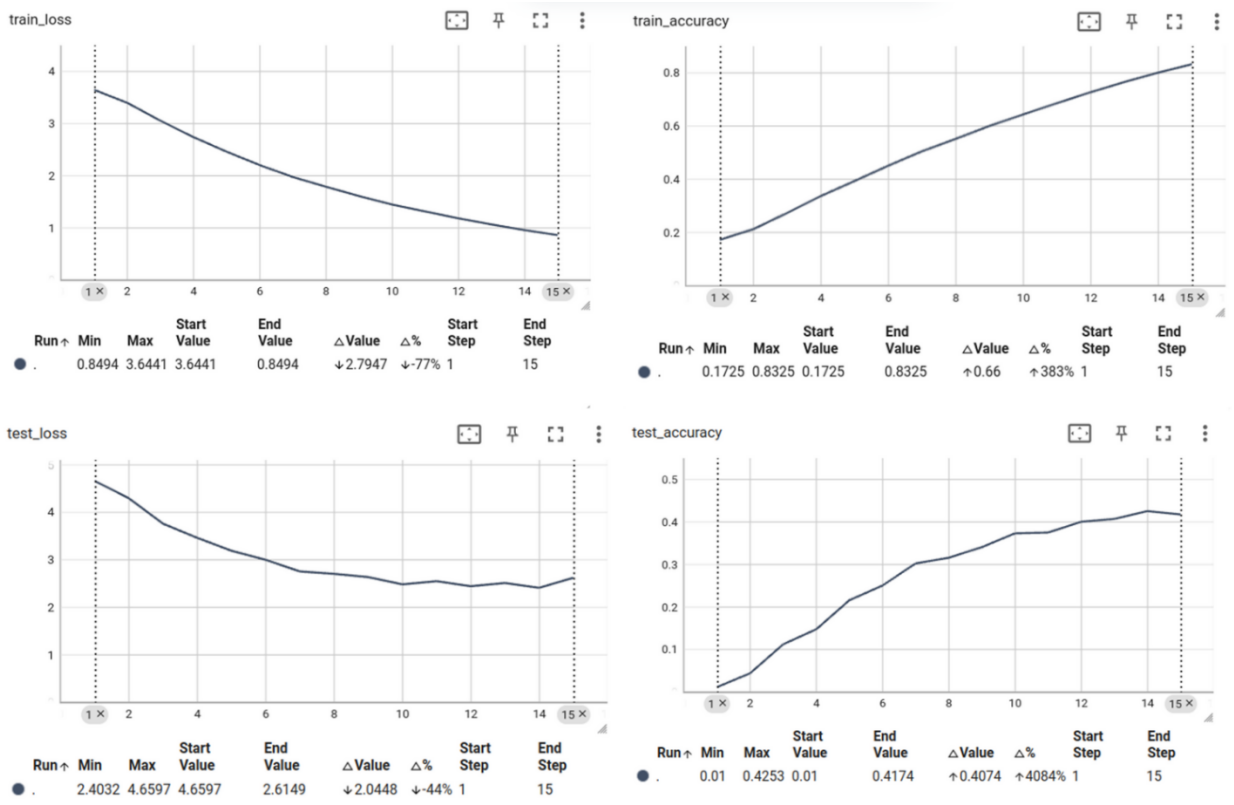


Figure 27. Training/test loss and accuracy

Power measurements are collected using the 'Delock WLAN Power Socket Switch MQTT with energy monitoring' switch, which sends data every 30 seconds. The entire training process lasted approximately 22 hours. Figure 28-31 show the power consumption over time for different devices. The Jetson AGX Orin consumes approximately 15 watts during model training, while the Jetson Nano with CPU uses about 7 watts, the Jetson Nano with GPU uses around 8 watts, and the Raspberry Pi uses roughly 6 watts.

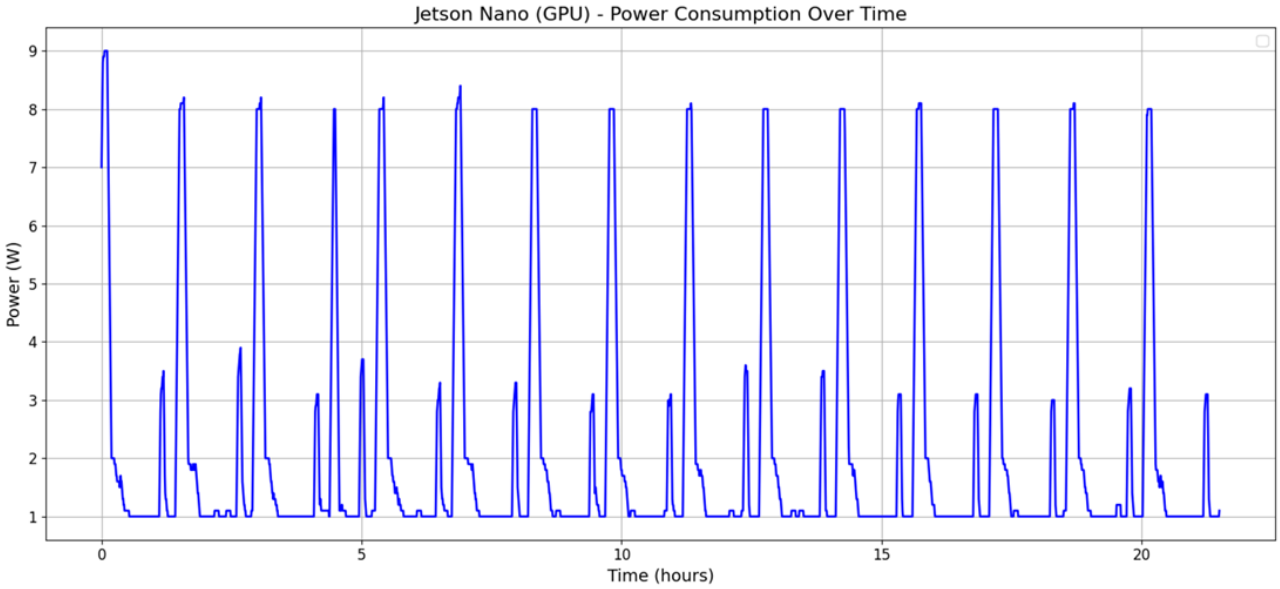


Figure 28. Jetson Nano utilizing GPU - Power consumption during training

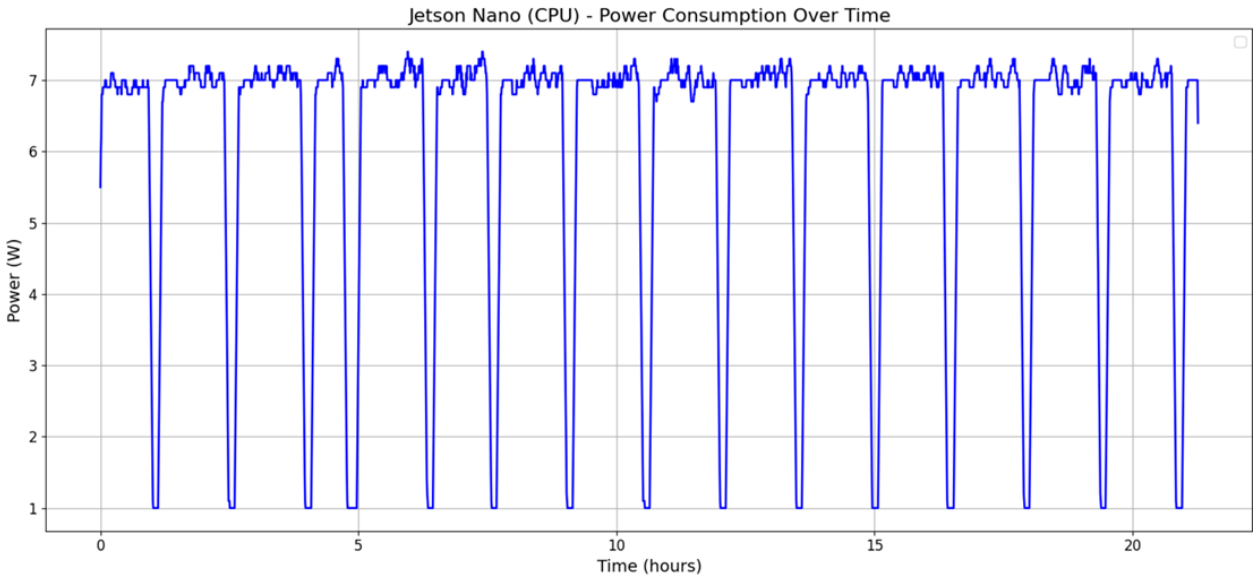


Figure 29. Jetson Nano utilizing CPU - Power consumption during training

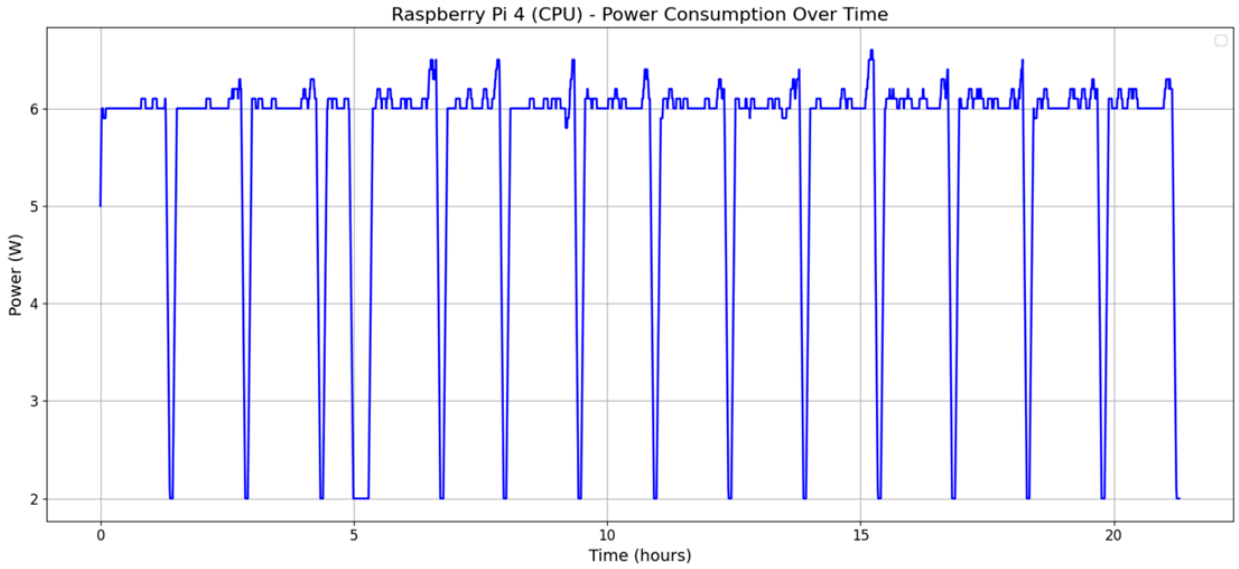


Figure 30. Raspberry Pi 4 utilizing CPU - Power consumption during training

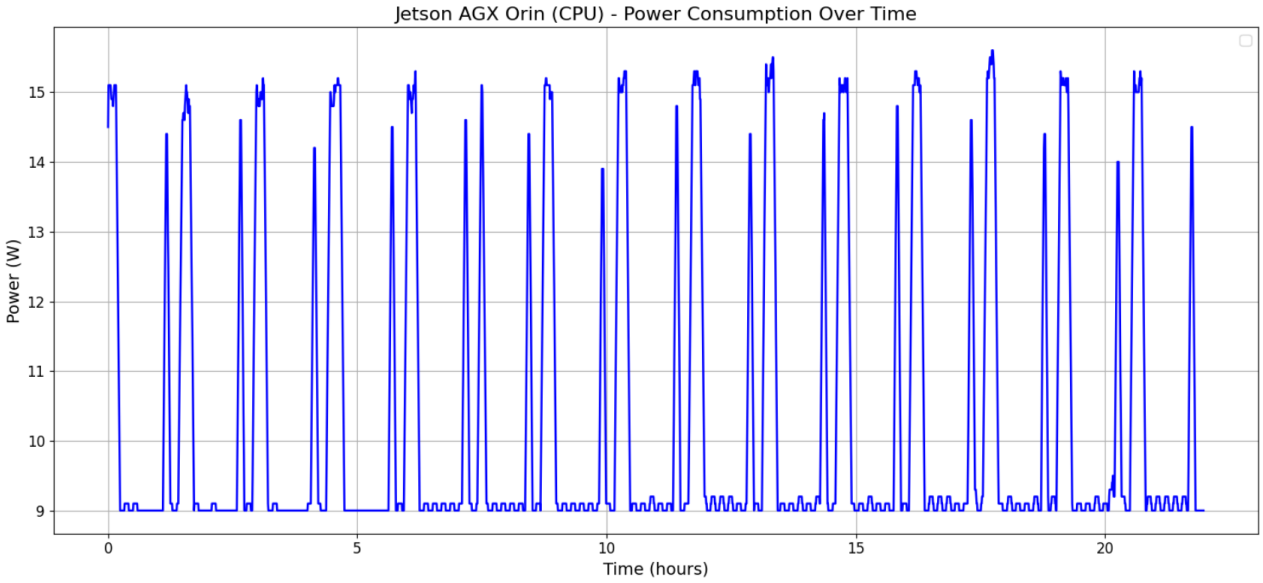


Figure 31. Jetson AGX Orin utilizing CPU - Power consumption during training

To calculate energy from the power data, it is assumed that power consumption remains constant throughout each 30-second interval. The energy consumed by the device during this period is measured using the following formula:

$$E = \frac{P \times \Delta t}{3600} [kWh]$$

Here, *P* represents the power measured during the interval, and Δ*t* is the duration of the interval, which is 30 seconds. Figure 32-35 display the energy consumption during training. Each line on the graph represents the energy consumption over a 2-minute interval, achieved by aggregating the energy consumption of four consecutive 30-second intervals to achieve clarity of the graphs. The Jetson AGX Orin consumed the most energy over time, approximately 227 kWh, followed by the Jetson Nano using the CPU at around 131 kWh, the Raspberry Pi 4 at around 120 kWh, and the Jetson Nano with the GPU, which consumed roughly 46 kWh of energy.

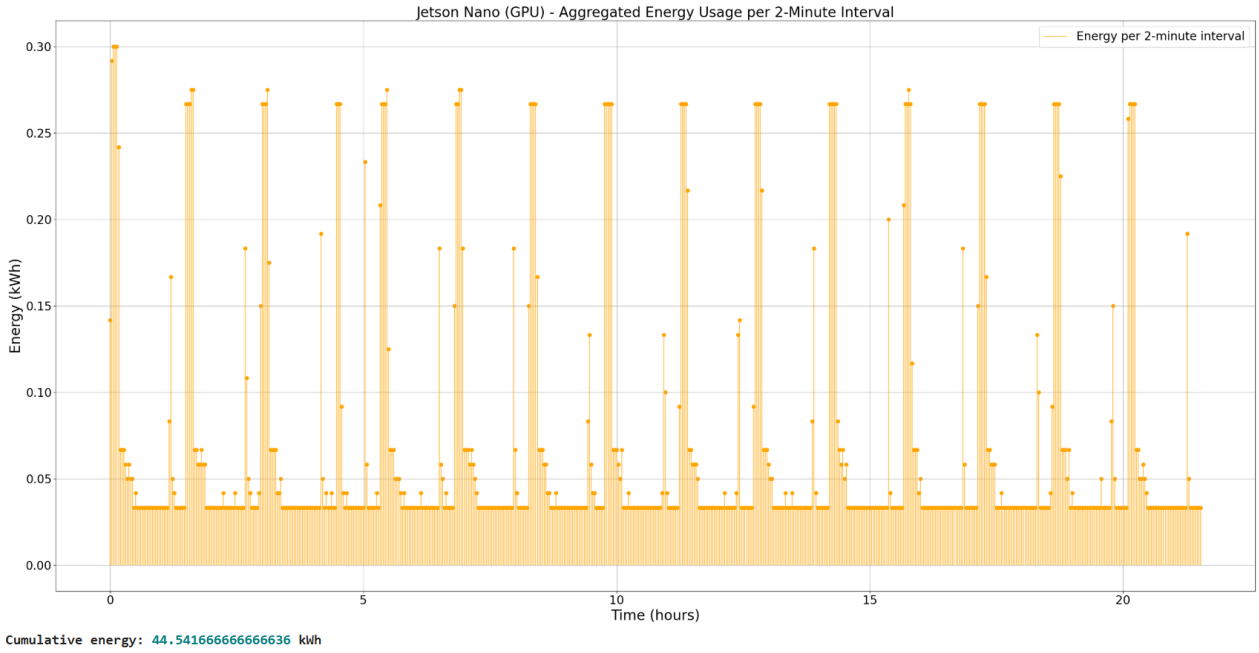
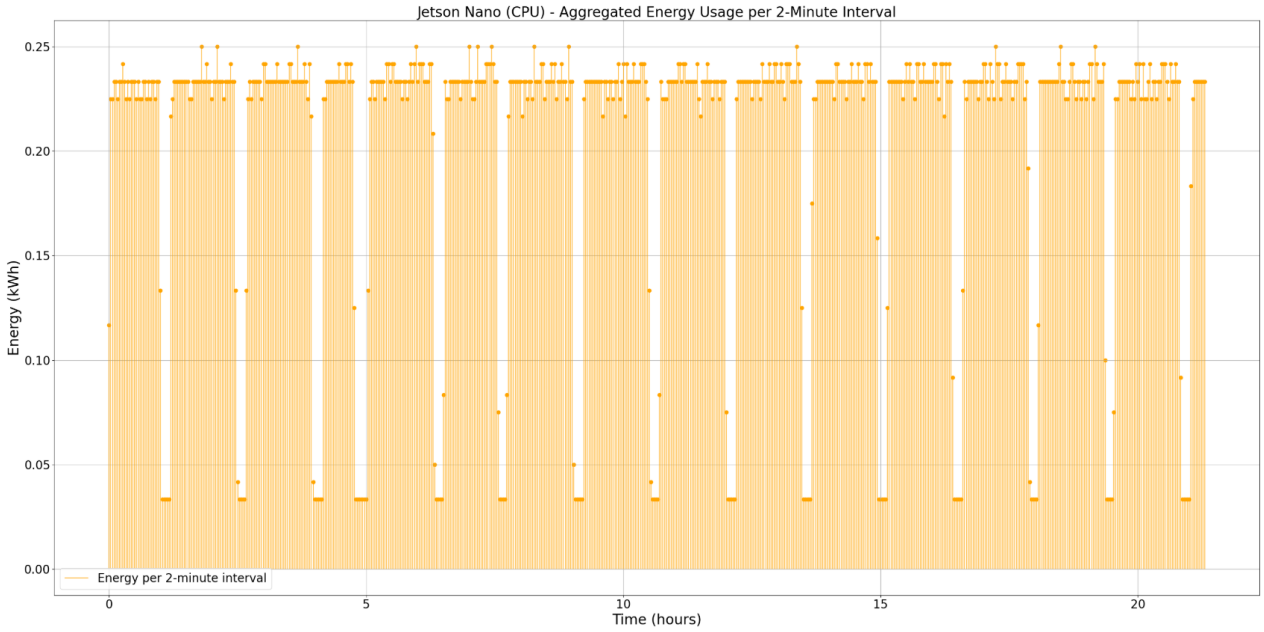
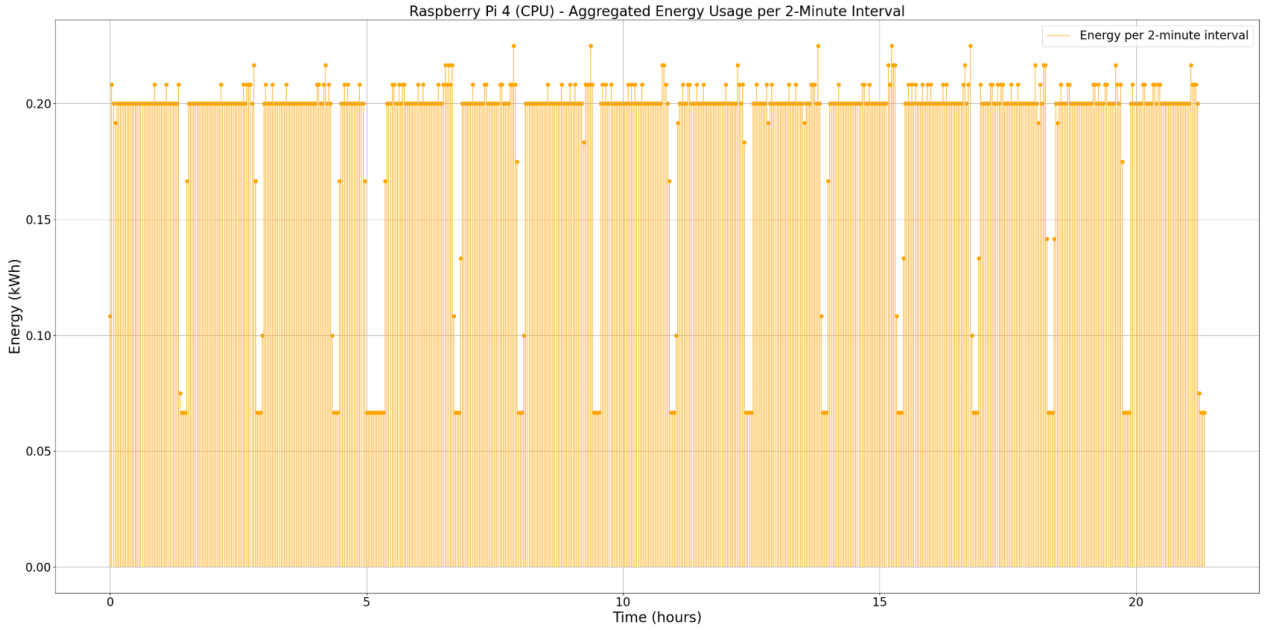


Figure 32. Jetson Nano utilizing GPU - Aggregated energy usage per 2-minute intervals during training



Cumulative energy: 131.46666666666616 kWh

Figure 33. Jetson Nano utilizing CPU - Aggregated energy usage per 2-minute intervals during training



Cumulative energy: 120.29166666666721 kWh

Figure 34. Raspberry Pi 4 utilizing CPU - Aggregated energy usage per 2-minute intervals during training

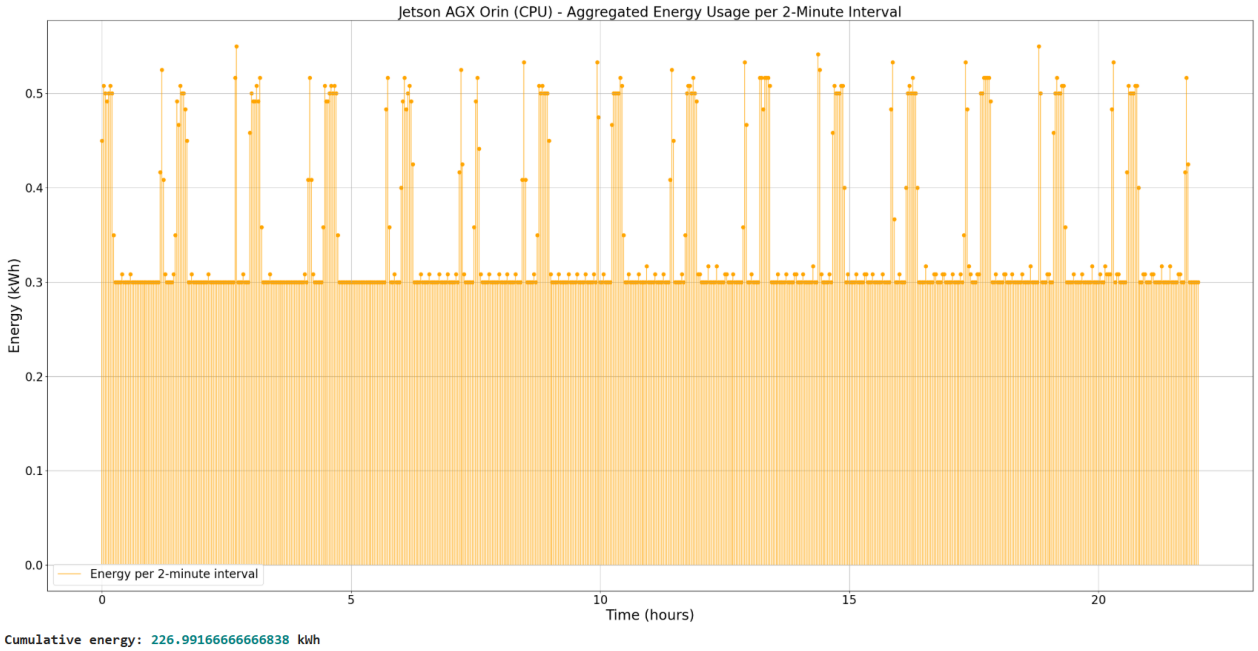


Figure 35. Jetson AGX Orin utilizing CPU - Aggregated energy usage per 2-minute intervals during training

4.2 WebAssembly runtime on Raspberry PI

This section analyses the performance of WebAssembly on Raspberry Pi devices. The analysis compares the performance metrics between the Raspberry Pi 4 Model B and Raspberry Pi 3 Model B+ in the execution of three projects, "gpio-http", "matrix" and “nn-module”. These tests, conducted with various WebAssembly runtime systems (Wasmtime, Wasmer, and WasmEdge), track parameters like CPU and memory usage, execution speed, file size, and energy consumption. Both Raspberry Pi devices use ARMv8-A 64-bit architecture processors, with differences in memory and processing power. The findings, shown in a series of tables, highlight how each runtime influences performance across different computational tasks and hardware setups, providing insights into efficiency and resource management for WebAssembly applications on edge devices.

The experiments were conducted on Raspberry Pi 4 Model B device, which feature an ARM Cortex-A72 processor [16], and Raspberry Pi 3 Model B+, which has an ARM Cortex-A53 processor [16]. The Raspberry Pi 4 used has 4GB of RAM, while the Raspberry Pi 3 has 1GB. During the execution of the projects, the following parameters were monitored:

- CPU usage,
- memory usage,
- file size,
- execution speed, and
- energy consumption.

File size was determined by summing the sizes of the binary file of the compiled Rust project and the size of the WebAssembly file. Execution speed was tracked using the Instant structure from Rust's standard

library, which is designed for time handling. CPU and memory utilization was monitored for two minutes during execution using the htop tool [18],. To collect data on electric power consumption, a Delock WLAN socket was used, which allows monitoring of electricity usage through a web interface.

The tables displaying the project testing results report the following parameters:

- Runtime - the execution system used to run the WebAssembly module;
- CPU - percentage of CPU utilization;
- RAM - percentage of memory utilized by the process;
- Current Intensity (mA) - average electric current intensity during execution, expressed in milliamper;
- Project (MB) - file size of the runtime that runs the module; and
- Module (KB) - size of the WebAssembly module.

To illustrate the impact of project execution, Table 9 and Table 10 show the parameters for Raspberry Pi 4 and Raspberry Pi 3 devices when no demanding processes are running.

Table 9. Parameters of the Raspberry Pi 4 device during idle period

CPU(%)	RAM(%)	Current (mA)
0	4	60

Table 10. Parameters of the Raspberry Pi 3 device during idle period

CPU (%)	RAM (%)	Current (mA)
0	26.6	60

4.2.1 Project gpio-http

This project accesses the device's hardware and sends an HTTP POST request to the server every two seconds. Parameters were monitored for approximately two minutes of execution. Execution data for the project is presented in Table 11 and Table 12.

Table 11. Parameters of the Raspberry Pi 4 device during execution of the gpio-http project

Runtime	CPU (%)	RAM (%)	Current (mA)	Project (MB)	Module (KB)
Wasmtime	11.8	0.5	61.46	16	35
Wasmer	11.0	0.6	63.30	29	35
WasmEdge	2.0	1.0	62.69	2.4	1300

Table 12. Parameters of the Raspberry Pi 3 device during execution of the gpio-http project

Runtime	CPU (%)	RAM (%)	Current (mA)	Project (MB)	Module (KB)
Wasmtime	28.2	2.2	62.90	16	35

Wasmer	30.6	2.4	63.10	29	35
WasmEdge	2.6	3.9	61.69	2.4	1300

The data shows that using the WasmEdge runtime results in significantly lower CPU utilization on both devices compared to other runtimes. This is because the communication with the server is handled within the WebAssembly module. However, there is a notable difference in the size of the WebAssembly module: WasmEdge's module occupies 1300 KB, whereas the modules used in the Wasmtime and Wasmer systems occupy only 35 KB. Memory usage and current intensity are approximately the same across all runtimes. Significant differences are evident in the project size, which is directly influenced by the execution system. WasmEdge is the most efficient with 2.4 MB, followed by Wasmtime with 16 MB, and Wasmer with 29 MB.

4.2.2 Project matrix

The matrix project runs a WebAssembly module that multiplies two 5000x5000 matrices to gather data during the execution of computationally intensive mathematical operations. Execution data for this project is shown in Table 13 and Table 14.

Table 13. Parameters of the Raspberry Pi 4 device during execution of the matrix project

Runtime	CPU (%)	RAM (%)	Current (mA)	Project (MB)	Module (KB)
Wasmtime	100	15.5	71.33	13	62
Wasmer	100	15.6	72.33	26	62
WasmEdge	100	15.1	71.33	2.2	129

Table 14. Parameters of the Raspberry Pi 3 device during execution of the matrix project

Runtime	CPU (%)	RAM (%)	Current (mA)	Project (MB)	Module (KB)
Wasmtime	100	50.2	73.12	13	62
Wasmer	100	50.4	73.91	26	62
WasmEdge	100	58.9	72.87	2.2	129

Matrix multiplication is a CPU-intensive task, and these results show the advantage of the newer processor in the Raspberry Pi 4 device. On this device, each project run resulted in 100% utilization of one of the four CPU cores and approximately 15% memory usage. The average current intensity was 71.33 mA for the Wasmtime and WasmEdge systems, and 72.33 mA for the Wasmer system, about 20% higher than when the device was idle.

On the Raspberry Pi 3 device, one CPU core was also fully utilized (100%), with memory usage reaching 50%. The current intensity was 73.12 mA for Wasmtime, 73.91 mA for Wasmer, and 72.87 mA for WasmEdge. The differences in project sizes are similar to the previous project, with WasmEdge being the

smallest at 2.2 MB, followed by Wasmtime at 13 MB, and Wasmer at 26 MB. Regarding the WebAssembly module size, the modules used in the Wasmtime and Wasmer systems occupy 62 KB, while WasmEdge runs an additionally optimized module of 129 KB.

For this project, the time required for matrix multiplication was also monitored, with data presented in Table 15. There were significant differences in execution times between devices. When using the Wasmtime runtime, the Raspberry Pi 3 requires 3.75 times more time for matrix calculations than Raspberry Pi 4, 4.7 times more with the Wasmer runtime, and 4.8 times more with WasmEdge. WasmEdge achieved the fastest matrix multiplication on Raspberry Pi 4, followed by Wasmer, and then Wasmtime. On the Raspberry Pi 3, WasmEdge was again the fastest, with Wasmtime in second and Wasmer in third place.

Table 15. Average time for multiplying 5000x5000 Matrices

Device	Runtime	Time (s)
Raspberry Pi 4	Wasmtime	152.749
Raspberry Pi 4	Wasmer	134.814
Raspberry Pi 4	WasmEdge	119.558
Raspberry Pi 3	Wasmtime	573.785
Raspberry Pi 3	Wasmer	634.586
Raspberry Pi 3	WasmEdge	572.085

4.2.3 Project nn-module

In this project, a TensorFlow Lite model was implemented to recognize bird species based on an image. The model loading and inference are done entirely within a WebAssembly module using the wasi-nn library, which currently contains experimental structures and interfaces for machine learning.

This project runs exclusively on the WasmEdge runtime. As shown in Table 16, running inference on a TensorFlow Lite model for bird species recognition is a demanding task for the CPU, though not as intensive for memory. The measured current intensity was 23% higher than when the device was idle. The newer processor's advantage is also evident here: inference time on the Raspberry Pi 4 is only 150 milliseconds, while on the Raspberry Pi 3 it takes 465.227 seconds which is a significant difference.

Table 16. Parameters for the nn-module project on Raspberry Pi devices

Device	CPU (%)	RAM (%)	Current (mA)	Module (KB)	Inference Time (s)
Raspberry Pi 4	100	3.2	74	997	0.150
Raspberry Pi 3	100	14.8	74	997	465.227

The results of these experiments underscore the impact of runtime choice and hardware differences on WebAssembly performance in edge computing environments. WasmEdge demonstrated lower CPU usage and smaller project sizes, making it well-suited for lightweight applications on both Raspberry Pi models. However, for CPU-intensive tasks, like matrix multiplication, all runtimes reached maximum CPU utilization, with the Raspberry Pi 4 offering better memory efficiency and lower current intensity than the Raspberry Pi 3. Overall, these findings highlight the importance of matching runtime environments with specific application needs to optimize resource usage on constrained devices.

5 Conclusion

This deliverable presents a demonstrator of state-of-the-art mechanisms for adaptive orchestration, energy efficiency, and machine learning (ML) workflows, explored through live experiments conducted in the relevant testbeds. The goal of this demonstrator was to assess and integrate mechanisms into a data-driven orchestration middleware for federated learning environments, enhancing the performance and energy efficiency of IoT systems supporting ML workflows.

The demonstrator was presented in three parts. The first part focused on optimizing federated learning pipelines through adaptive orchestration, where the development of the Reconfiguration Validation Algorithm (RVA) and its associated framework demonstrated significant improvements in the dynamic reconfiguration and flexibility of FL workflows. The second part explored the application of spatio-temporal graph neural networks and gossip learning for decentralized FL setups. Experimental results compared different training configurations and evaluated cloudlet performance, highlighting the benefits of decentralized learning in reducing server dependency.

The third part combined the evaluation of energy consumption and performance optimization in FL and WebAssembly runtime environments on edge devices. Experiments conducted on devices such as Jetson AGX Orin, Jetson Nano, and Raspberry Pi 4 highlighted the energy efficiency of each device in FL training scenarios, while performance tests on WebAssembly runtimes demonstrated resource usage and efficiency in computational tasks like ML inference and matrix multiplication. The findings provide valuable insights into optimizing energy use and computational efficiency in resource-constrained and edge environments.

6 Acronyms

AI	Artificial Intelligence
AIoT	Artificial Intelligence of Things
CC	Computing Continuum
FL	Federated Learning
GNNs	Graph Neural Networks
IID	Independent and Identically Distributed
IoT	Internet of Things
non-IID	non-Independent and Identically Distributed
MQTT	Message Queuing Telemetry Transport
QoS	Quality of Service

7 List of Figures

8 List of Tables

9 References

[1] Čilić, I., Lackinger, A., Frangoudis, P., Žarko, I. P., Furutanpey, A., Murturi, I., & Dustdar, S. (2024). Reactive Orchestration for Hierarchical Federated Learning Under a Communication Cost Budget. ArXiv. https://arxiv.org/abs/2412.03385

[2] Cloud Native Computing Foundation, "K3s - lightweight kubernetes," https://docs.k3s.io/

[3] Zagoruyko, Sergey. "Wide residual networks." arXiv preprint arXiv:1605.07146 (2016).

[4] Wu, Zonghan, et al. "A comprehensive survey on graph neural networks." IEEE transactions on neural networks and learning systems 32.1 (2020): 4-24.

[5] Sahili, Zahraa Al, and Mariette Awad. "Spatio-temporal graph neural networks: A survey." *arXiv preprint arXiv:2301.10569* (2023).

[6] Ruiz, Luana, Fernando Gama, and Alejandro Ribeiro. "Gated graph recurrent neural networks." *IEEE Transactions on Signal Processing* 68 (2020): 6303-6318.

[7] Yu, Bing, Haoteng Yin, and Zhanxing Zhu. "Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting." *arXiv preprint arXiv:1709.04875* (2017).

[8] Wu, Tianlong, Feng Chen, and Yun Wan. "Graph attention LSTM network: A new model for traffic flow forecasting." *2018 5th international conference on information science and control engineering (ICISCE)*. IEEE, 2018.

[9] Giaretta, Lodovico, and Sarunas Girdzijauskas. "Fully-Decentralized Training of GNNs using Layer-wise Self-Supervision." (2023).

[10] Nazzal, Mahmoud, et al. "Semi-decentralized inference in heterogeneous graph neural networks for traffic demand forecasting: An edge-computing approach." *IEEE Transactions on Vehicular Technology* (2024).

[11] Kralj, Ivan, et. al. "Semi-decentralized Training of Spatio-Temporal Graph Neural Networks for Traffic Prediction" *arXiv preprint arXiv: 2412.03188* (2024).

[12] Kairouz, Peter, et al. "Advances and open problems in federated learning." *Foundations and trends® in machine learning* 14.1–2 (2021): 1-210.

[13] He, C., et al. "Central server free federated learning over single-sided trust social networks. arXiv 2019." *arXiv preprint arXiv:1910.04956*.

[14] Ormándi, Róbert, István Hegedűs, and Márk Jelasity. "Gossip learning with linear models on fully distributed data." *Concurrency and Computation: Practice and Experience* 25.4 (2013): 556-571.

[15] Defferrard, Michaël, Xavier Bresson, and Pierre Vandergheynst. "Convolutional neural networks on graphs with fast localized spectral filtering." *Advances in neural information processing systems* 29 (2016).

[16] Arm, "Cortex-a72", https://www.arm.com/products/silicon-ip-cpu/cortex-a/ cortex-a72

[17] arm Developer, "Arm cortex-a series programmer's guide for armv8-a", https: //developer.arm.com/documentation/den0024/a/ARMv8-A-Architecture-andProcessors/ARMv8-A

[18] htop.dev, "htop", https://htop.dev

