Twinning action for spreading excellence in Artificial Intelligence of Things

DELIVERABLE D1.2

DEMONSTRATOR OF STATE-OF-THE-ART 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.	**D1.2**
Title	Demonstrator of State-of-the-Art Mechanisms for Edge Orchestration, Energy-Efficiency and ML Workflows
Lead Editor	Ivan Čilić
Type	Report
Dissemination Level	PU/SEN
Work Package	WP1
Due Date	28.02.2024

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
Lodovico Giaretta	RISE	lodovico.giaretta@ri.se

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
0.01	18-03-2024	Mario Kušek	Initial structure
0.02	27-03-2024	Ivan Čilić	Performance orchestration of container orchestration tools: environment and evaluation
0.03	28-03-2024	Mislav Has	Power consumption experiment setup and device states used for measurement
0.04	29-03-2024	Ivan Čilić	Edge orchestration experiments with emulated network: environment and evaluation
0.05	30-03-2024	Mislav Has	Power consumption results
0.07	30-03-2024	Ana Petra Jukić	Federated learning on heterogeneous nodes with dataset from VisionKG: software and results
0.08	30-03-2024	Katarina Vuknić	Federated learning on heterogeneous nodes with dataset from VisionKG: hardware, software and results
0.09	04-04-2024	Ivan Čilić	Introduction, Executive Summary and Conclusion
0.10	05-04-2024	Mario Kušek	Internal review

0.11	06-04-2024	Ivan Čilić	Version sent to Lodovico Giaretta for internal review
0.12	12-04-2024	Lodovico Giaretta	Reviewing and writing comments and suggestions
0.13	22-04-2024	Mislav Has, Ivan Čilić, Ana Petra Jukić, Katarina Vuknić	Content finalization based on internal review
1.00	30-04-2024		Final version submitted to EC

Table of Contents

Executive Summary

This deliverable presents the demonstrator of state-of-the-art mechanisms for edge orchestration, energy-efficiency and ML workflows. The demonstrator was implemented in three parts tackling (1) service orchestration at the edge, (2) power consumption of resource-constrained devices, and (3) federated learning. To demonstrate service orchestration at the edge, a performance evaluation of container orchestration tools for edge environments was conducted. Based on the evaluation results, an experiment with K3s was conducted to demonstrate how to run edge experiments in an emulated network. The second part demonstrates the measurement of power consumption on resource-constrained devices using an STM32 device. A model containing several different device states was created to measure the power consumption of the actions that a device commonly performs in an IoT use case. Finally, the third part of the demonstrator shows the federated learning deployment on a set of heterogeneous edge devices and its execution with a dataset obtained from the VisionKG framework [12]. The deployment involves training a CNN in a federated learning setup using a MNIST dataset of handwritten digits.

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.

This research is related to the following three AIoTwin research domains:

- Orchestration in the edge-to-cloud continuum,
- Federated and decentralised learning and
- Robust and energy efficient IoT.

In this work package, our objective is to develop an edge orchestration middleware that places ML models within the edge-to-cloud continuum, considering data streams from diverse IoT devices. Additionally, the middleware takes into account the energy efficiency of edge orchestration deployments that support ML workflows. To lay a solid foundation for this middleware, this report showcases current state-of-the-art mechanisms for orchestration, energy efficiency, and ML workflows tailored for IoT edge environments on our established testbed. Through live experiments conducted on the UNIZG-FER testbed, the demonstrator underscores the importance of service orchestration at the edge and demonstrates mechanisms for edge orchestration, energy monitoring, and federated learning. This demonstrator exhibits components that will be integrated in middleware defined in deliverable D1.1.

1.2 Deliverable Outline

This document is organized as follows. Section 2 evaluates different container orchestration tools for the edge environments and describes running experiments for edge orchestration in an emulated network. Section 3 shows the setup and results of measuring power consumption on a resource-constrained STM32 device. Section 4 demonstrates a federated learning deployment on heterogeneous nodes training on a dataset obtained from VisionKG. Section 5 provides the conclusion of the deliverable. A list of acronyms, figures, tables and references can be found at the end of the document.

2 Service orchestration at the edge

Edge computing aims to improve quality of service (QoS) for IoT devices and application-level services in an Edge-to-Cloud Continuum (ECC) by deploying cloud functions closer to the network edge. To maintain optimal QoS, service orchestration mechanisms are crucial for deploying services to suitable edge nodes, monitoring service performance and migrating to another edge nodes as needed. This section describes the ECC service orchestration architecture proposed in [1] and its key components.

There are four main components that form the ECC service orchestration architecture:

- IoT device
- Edge node
- Edge service
- Orchestrator

Edge computing is primarily focused on IoT devices with limited resources, which typically perform tasks such as environmental sensing, data publishing, and actuation. In the proposed architecture, IoT devices connect to the Internet via gateways, necessitating additional computing nodes for data processing or actuation commands. Edge services handle data processing and sending actuation commands to the IoT devices. An edge service assigned to an IoT device must be autonomous, stateless, and portable to facilitate quick migrations and ensure high availability. Container virtualization is a preferred approach for running edge services due to its portability and reduced start-up times. Edge nodes, organized in a single or multi-layer network, host containerized edge services and are ideally placed at the far edge layer for proximity to IoT devices. The orchestrator, central component of the architecture, has the role of discovering and managing edge nodes, and deploying services to the optimal nodes with respect to service QoS requirements. It can be centralized or distributed across the edge environment, with functionalities placed in the cloud or on different edge layers based on use case requirements.

2.1 Performance evaluation of container orchestration tools for the edge

In this section, we provide the comparison of the most popular container orchestration tools applicable for service orchestration at the edge: Kubernetes, K3s, KubeEdge and ioFog [2]. The comparison was made in two phases: deployment evaluation (qualitative) and performance evaluation (quantitative). In the deployment evaluation, we evaluate the complexity of deployment and resources used by the tool, and in the performance evaluation we evaluate startup and migration time for a selected service.

2.1.1 Environment setup

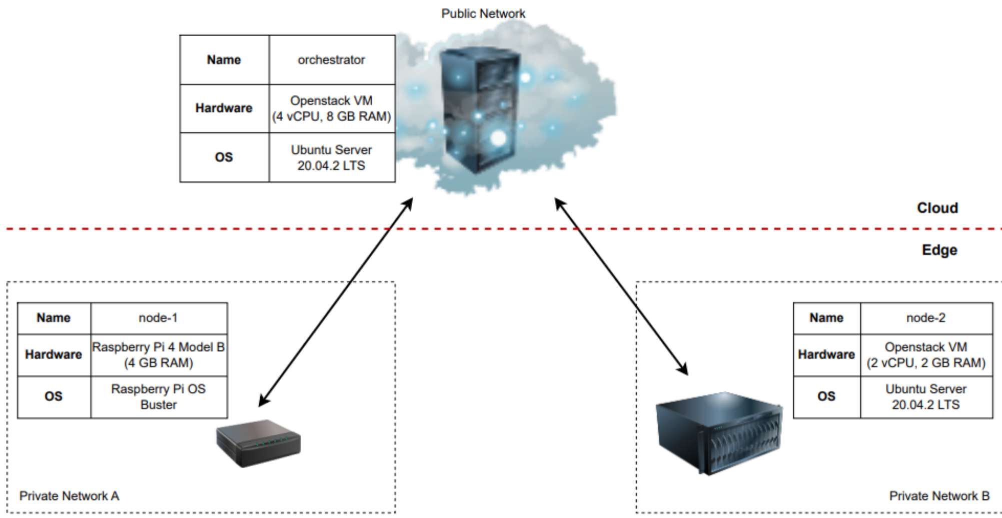

Figure 1 Configuration of the evaluation environment.

Figure 1 illustrates the environment setup designed for evaluating orchestration tools, providing hardware and software specifications. The evaluation utilized three nodes: one for hosting the ECC controller and two as edge nodes. This setup aims to address the following questions:

- How does the orchestration tool handle edge nodes in private networks?
  - Edge nodes were placed in different private networks, and the controller was placed in the public network.
- How does the orchestration tool support node heterogeneity?
  - Nodes were deployed on both x86 and ARM processor architectures.
- How does the orchestration tool run on a resource-constrained edge node?
  - Node *node-1* runs on a single-board Raspberry Pi 4 computer.

2.1.2 Cluster deployment evaluation

The initial phase of the evaluation focused on deploying the cluster using the chosen tool, aiming to qualitatively assess various aspects such as deployment complexity, support for different node platforms, and the memory footprint on edge nodes. Deployment of the cluster involved the following steps for each tool:

1. Reading and evaluating the tool documentation, focusing on cluster deployment instructions.
2. Installing and setting up the prerequisites for cluster deployment.
3. Deploying the cluster with the orchestrator running on the cloud node *orchestrator*.
4. Adding worker node *node-1* to the cluster and measuring its RAM consumption. Removing the node after the measurement and repeating this step several times to obtain an average value.
5. Adding worker node *node-2*.
6. Deploying our edge service on both nodes and connecting to it to verify that the cluster is working.

Table 1 Cluster deployment comparison.

Parameter	**Kubernetes**	**K3s**	**KubeEdge**	**ioFog**
Documentation Quality	High	Medium	Medium	Low
Deployment Complexity	High	Low	Medium	Medium
Private Networks	No	Yes	Yes	Yes
Heterogeneity	Yes	Yes	Yes	Yes
Resource-Constrained	Yes	Yes	Yes	Yes
Memory Footprint (MB)	~50	~50	~40	~240

Table 1 displays a comparison of selected container orchestration tools based on cluster deployment. Kubernetes emerges as the leader in documentation quality, benefiting from a large community and industry standard status. On the other hand, ioFog's documentation lacked clarity regarding supported OS releases, resulting in deployment issues. Deploying a Kubernetes cluster proved to be the most time-consuming due to the absence of built-in support for nodes in different subnets, necessitating the setup of a VPN network, such as Wireguard. KubeEdge deployment requires setting up a Kubernetes control-plane node which adds some complexity, while ioFog requires SSH connections to each node with *sudo* permissions. K3s stands out as the simplest to deploy, requiring only one command for controller provision and one command for node joining. All tools support nodes running on both x86 and ARM architectures, with no significant performance degradation observed on an SBC like the Raspberry Pi 4.

SBCs are commonly considered for edge nodes due to their low power consumption and capability to handle high-performance tasks. Memory consumption analysis on node *node-1* running on a Raspberry Pi 4 gave unexpected results, challenging the perception of Kubernetes as *„too heavy“* to run at the edge. The memory overhead of running a Kubernetes worker node on a Raspberry Pi is comparable to K3s or KubeEdge, despite K3s being known as *„lightweight Kubernetes“* and KubeEdge being tailored for edge use. Finally, ioFog exhibits significantly higher memory footprint, nearly five times greater than the other tools.

2.1.3 Performance evaluation

After deploying the cluster, we were able to quantitatively evaluate the tools by measuring the time required for deploying and migrating a specific service. To conduct this assessment, we developed the service as a Node.js HTTP server, responding to GET requests on the root path. The server was containerized as a Docker image and uploaded to the Docker Hub public container image registry.

Table 2 Performance comparison results

Time (s)	**Kubernetes**	**K3s**	**KubeEdge**	**ioFog**
Total Startup Time	1.799	2.798	2.858	34.537
Total Migration Time	1.838	2.782	2.781	23.244
Startup Time Overhead	0.504	1.502	1.5622	33.326
Migration Time Overhead	0.542	1.486	1.485	22.032

Table 2 shows the performance evaluation results by providing the average number of total service startup and migration times for each of the four container orchestration tools used in this study. The experiment was repeated 10 times to obtain the average number of times needed for service startup and migration. The table also shows the overhead of a tool, which was determined by subtracting the container engine startup time from the total time. Kubernetes, K3s and KubeEdge were running *containerd* which took on average 1.295 seconds to start the given service and ioFog was running Docker Engine which needed 1.211 seconds.

ioFog exhibits the poorest performance in terms of service startup and migration time, with a significant delay of about 30 seconds observed in initiating container startup requests, likely due to prolonged request processing time at the controller. This issue, previously identified in our research [1], remains unresolved, which makes ioFog unsuitable for on-demand edge service deployment. K3s and KubeEdge demonstrate comparable performance, both taking approximately 2.8 seconds for service to be deployed and available. In contrast, Kubernetes exhibits the fastest startup time at around 1.8 seconds. This disparity in performance can be attributed to additional edge-specific components in KubeEdge, such as CloudCore and EdgeCore, which introduce overhead in metadata storage and retrieval. Meanwhile, K3s utilizes lightweight SQLite storage, potentially impacting performance compared to Kubernetes with etcd. Migration times are more or less comparable to the service startup time, meaning there is no significant overhead when performing migrations. This is important in edge environments with unstable nodes and networks where migrations occur with high frequency.

2.2 Conducting edge orchestration experiments with an emulated network

Given the dynamic and distributed nature of the ECC environment, employing a network simulation tool becomes essential to accurately evaluate edge orchestration solutions. For example, if we want to test our service placement algorithm for ECC in a real deployment scenario, two key components are required: a service orchestration tool for deploying and managing services across distributed nodes, and a network simulator or emulator to mimic network failures and delays. Therefore, in this section, we describe the process of conducting experiments on edge orchestration solutions within an emulated network environment. Specifically, we demonstrate how to execute experiments by deploying a K3s cluster atop an emulated network using Imunes.

Imunes is an integrated multiprotocol network emulator/simulator based on the FreeBSD and Linux operating system kernel, partitioned into several lightweight virtual nodes that can be interconnected via kernel-level links to form arbitrarily complex network topologies [3]. Imunes can be used to simulate network delays and failures between ECC nodes and devices with the option to emulate the devices as Docker containers.

Based on the comparison in Section 2.1, we have decided to use K3s as the edge orchestration tool. The main reason to use K3s is its deployment simplicity and the fact that it relies on Kubernetes, a stable and powerful container orchestration tool.

2.2.1 Environment setup

The experiment was designed to follow the example of testing a service placement algorithm for ECC aimed at minimizing request latency. For this purpose, the experimental setup consists of distributed nodes hosting service replicas, along with a client responsible for sending requests to these replicas deployed across various nodes. This configuration enables evaluation of the placement algorithm's effectiveness in optimizing request latency across the distributed environment.

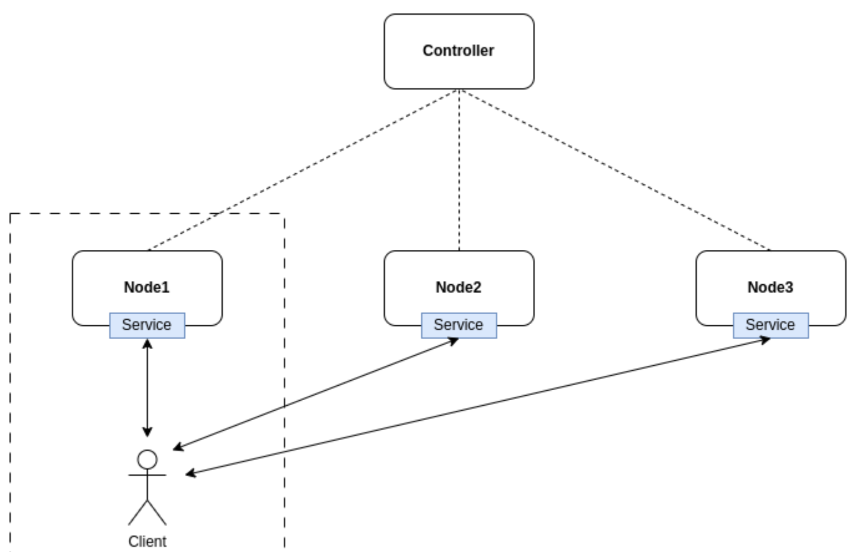

Figure 2 Environment setup for edge orchestration experiment.

Figure 2 shows the experimental environment consisting of four nodes: a controller node of K3s and three worker nodes. Each node runs one replica of a service, i.e. service instance, and a client, which is in the same local network as the node *Node 1,* sends requests to all replicas to measure the request latency.

To implement such an environment and integrate it with an emulated network, the following steps are required:

1. *Create virtual machines (VMs) to serve as edge nodes.* We decided to use VMs to have full control over resource utilization and ensuring uniformity among nodes. Alternatively, containers or physical hardware can be utilized, which can be integrated into the Imunes network.
2. *Add interfaces of VMs to Imunes and configure the network.* Imunes allows attaching external interfaces to the network, so this step only requires adding the interface of newly created VMs. After that, routers, switches and links need to be configured between VM interfaces to simulate network delays and failures.
3. *Configure routes in VMs.* In a newly created network, IP routes need to be configured in VMs to communicate with VMs in different subnets.
4. *Deploy K3s cluster.* Once the network configuration is complete and all the VMs are interconnected, a K3s cluster can be deployed. With only two commands the controller is started and the worker nodes join the cluster by specifying the controller's address and the node join token.

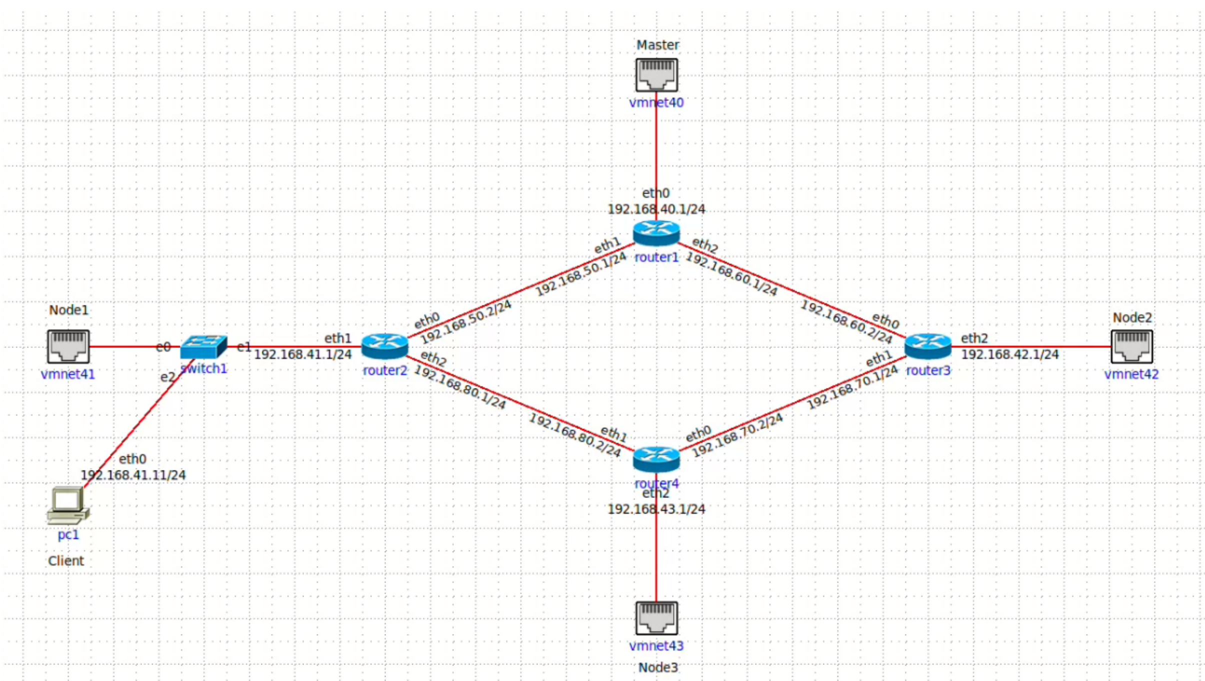

Figure 3 Experiment visualization in Imunes.

Figure 3 shows how the experiment is configured in Imunes. We can observe that the nodes are distributed in different networks and that the client is running in the same network as *Node 1*.

2.2.2 Experimental evaluation

Once the setup is complete, we deploy the services and start the experiment. The evaluation was conducted running a simple HTTP service that upon received request returns a response to the client on the successful processing of request data.

```
Every 2.0s: kubectl get pods -o wide

NAME                                 READY   STATUS    RESTARTS   AGE   IP          NODE    NOMINATED NODE   READINESS GATES
edge-deployment1-55974c5b55-9xd55    1/1     Running   0          54s   10.42.1.3   node1   <none>           <none>
edge-deployment2-5ddfddd6bb-8l8zm    1/1     Running   0          51s   10.42.2.3   node2   <none>           <none>
edge-deployment3-6c5cb75489-hflfb    1/1     Running   0          48s   10.42.4.3   node3   <none>           <none>
```

Figure 4 Service instances deployed in K3s.

We can observe in Figure 4 that all three service instances were successfully started. After that, we start the client. The client was simulated in a bash script that sends 10 requests to a service instance URL received as an input by the user. After every 10 requests, the user can enter a different service instance URL. The script measures the request latency and stores it to an output file.

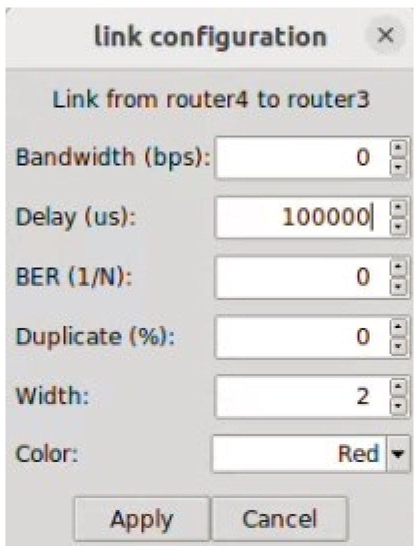

Figure 5 Link configuration in Imunes.

We simulated delays by changing delay on links between client and the nodes. Figure 5 shows how a link delay is set to 100 milliseconds in Imunes.


```
Request number: 27
Relative Time: 1913 ms
Service URL: 192.168.42.10:30002
Request Duration: 15 ms
Hello from the edge service!
---------------------------------------------------
Request number: 28
Relative Time: 1982 ms
Service URL: 192.168.42.10:30002
Request Duration: 13 ms
Hello from the edge service!
---------------------------------------------------
Request number: 29
Relative Time: 2046 ms
Service URL: 192.168.42.10:30002
Request Duration: 9 ms
Hello from the edge service!
New endpoint:
192.168.42.10:30002
Script resume time (3)
2023-09-04T19:47:32.589Z
---------------------------------------------------
Request number: 30
Relative Time: 2284 ms
Service URL: 192.168.42.10:30002
Request Duration: 235 ms
Hello from the edge service!
---------------------------------------------------
Request number: 31
Relative Time: 2585 ms
Service URL: 192.168.42.10:30002
Request Duration: 245 ms
Hello from the edge service!
```

Figure 6 Change in request latency after link delay change.

Finally, in Figure 6 we can observe the change in request latency obtained from the output file created during the experiment. The figure shows the latency towards the service running on *Node 2* before and after a delay of 100 milliseconds was introduced on the link between the routers that connect the client and the node. It can be observed that before introducing the delay the latency was approximately 10 milliseconds and after the delay it was approximately 240 milliseconds due to the round-trip delay of 200 milliseconds.

The experiment highlighted the ease of validating edge orchestration solutions in an environment closely replicating actual deployment conditions. Conducting experiments in an emulated network on a real cluster allows for a comprehensive assessment of how infrastructure impacts solution functionalities, which is difficult to accomplish using simulation tools alone.

3 Measuring power consumption of a resource-constrained device

For the demonstrator, we measured the power consumption on the STM32 device. The main goal is to identify the main power consumers in relation to the total consumption of the device for a given use case. We created a model that contains several different states of the device where we track the transitions of the device from one state to another. The states we used represent typical states that are commonly used on a resource-constrained devices in IoT environment. Marking the state helps us to measure the power consumption of the actions the device performs. To recognise which state the device is in, we used GPIO pins on the STM32 device. We have used 3 pins that change their values from logic 0 to 1 to represent the states from 0 to 6.

3.1 Experiment setup

For testing, we used equipment from Mikroelektronika [4]. The device is a UNI Clicker [5] with MCU card STM32F767ZG. The sensors used are Environment click [6] with a BME680 chip, which measures temperature, humidity and pressure. The LTE IOT 9 click [7] was used as the communication module, which enables connections to the LTE CAT-M1 and CAT NB1/2 network. We used NB-IoT technology to connect to the Internet and send data with Message Queuing Telemetry Transport (MQTT) protocol [9].

For power measurement we used the Power Profiler Kit II (PPK2) manufactured by Nordic Semiconductor [8]. The Power Profiler Kit II (PPK2) is a standalone device that can measure and optionally deliver currents from sub-μA up to 1A. The PPK2 can also use digital inputs as a low-end logic analyser, enabling code-synchronised measurements. For this purpose, the digital inputs are connected to an I/O pin of the external device under test, in our case STM32. The experimental setup with the described equipment can be seen on Figure 7.

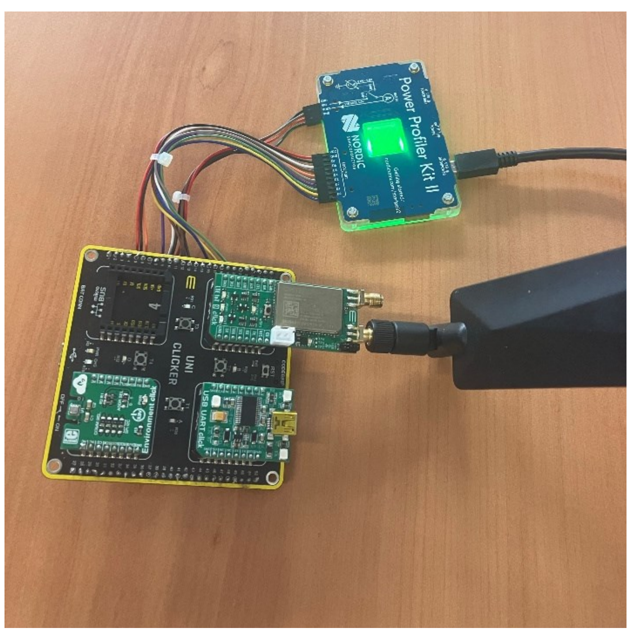

Figure 7. Experiment setup for power measurement

Through its interface, PPK2 displays the results in a diagram that shows the current over time. It indicates the current generated by the device during a certain moment. The device operates at a constant voltage of 3.3 V. In order to obtain the power consumption under certain conditions, the following formula must be calculated:

$$P = U \times I$$

3.2 Device states used for measurements

The defined states and their corresponding symbolic representations are stated in Table 3.

Table 3. Device states for power consumption measuring

WAKE_UP_DEVICE_AND_MODULE	0
CONFIGURATION_FOR_NETWORK	1
CHECK_NETWORK_CONNECTION	2
SENSOR_MEASUREMENT	3
SEND_MESSAGE	4
RECEIVE_MESSAGE	5
END_CONNECTION	6

WAKE_UP_DEVICE_AND_MODULE (0):

This initial state includes the process of waking up the device, as well as initializing the communication module to make it ready for subsequent operations. It serves as a preparatory phase before the transition to the CONFIGURATION_FOR_NETWORK status.

CONFIGURATION_FOR_NETWORK (1):

This state includes activities related to the configuration of network settings to ensure that the optimal connectivity parameters are set up for subsequent operations.

CHECK_NETWORK_CONNECTION (2):

In this state, the device actively checks its network connection and evaluates the integrity and availability of the network infrastructure. The device remains in this state until the connection is established.

MEASUREMENT (3):

In the measurement state, the device is busy capturing and processing sensor data to provide insightful measurements for subsequent analysis.

SEND_MESSAGE (4):

The state in which the parameters required to establish an MQTT connection are defined and the message is defined and sent.

RECEIVE_MESSAGE (5):

In contrast to the "send message" state, the device in this state is configured to receive incoming data or messages and therefore offers bidirectional communication options.

END_CONNECTION (6):

This state indicates the completion of network-related activities and includes tasks such as closing network connections or releasing allocated resources after operations have been completed.

3.3 Results

The primary objective was to identify and address the key power consumers within different operational contexts. Our approach encompassed the measurement of power consumption across distinct states, each representing a specific operational scenario. As the device moved from one state to another, the power consumption changed. In the results, we highlight the main power consumers that we have identified in individual states. In this context, we isolated the key stages of the device:

Active State with Communication Module Off:

This state represents the device's operation with the communication module deactivated. It serves as a baseline for power consumption when the device is in an active state but not engaged in communication tasks. Since the communication module starts up at the beginning of our use case, this state has a short duration. In that period, it was found that the device consumes an average current of 126.86 mA.

Active State with Communication Module On:

Contrasting with the previous state, here the communication module is actively engaged. This state enables us to assess the incremental power consumption attributed to communication activities. Turning on the communication module leads to a sudden increase in the amount of current generated. With an average of 126.86 mA before turning on, and peak of 403.92 mA, the average current comes to the amount of 228.25 mA with the module on. The moment at which the module is switched on and the average values before and after this can be seen in Figure 8.

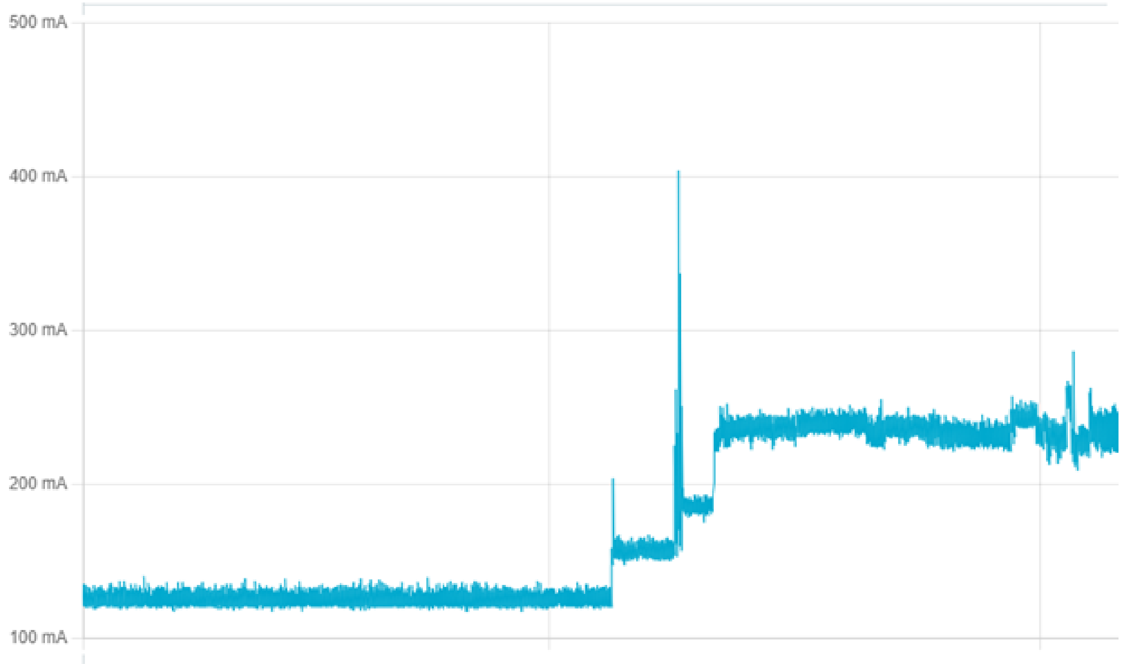

Figure 8. Current consumption when communication module is turned on

Communication with the module then takes place via AT commands [10], whereby the configuration settings for the module are defined and the module is connected to the Internet. During this period, the device generates an average current of **251.49 mA**.

Sensor Measurements:

In this state, the device is focused on sensor data acquisition. It allows us to get the power consumption associated with sensor interfacing and data acquisition processes. During this period, three values are retrieved from the sensor: temperature, humidity and pressure. In this state, the average current consumed is 192.68 mA. It should be noted that while the communication module is still active, there is no interaction with it during this period, only with the sensor module.

Sending Data:

During data transmission, the device is actively involved in sending data to external receivers. This state facilitates the examination of power consumption patterns during data transmission operations. Before the actual sending process, the configuration parameters for the MQTT connection are defined, also via AT commands. The relationship between the period in which the MQTT parameters were defined and the moment in which the connection was established and the message was sent can be seen in Figure 9. The amount of current jumped from an average of 178.36 mA to an average of 248.44 mA.

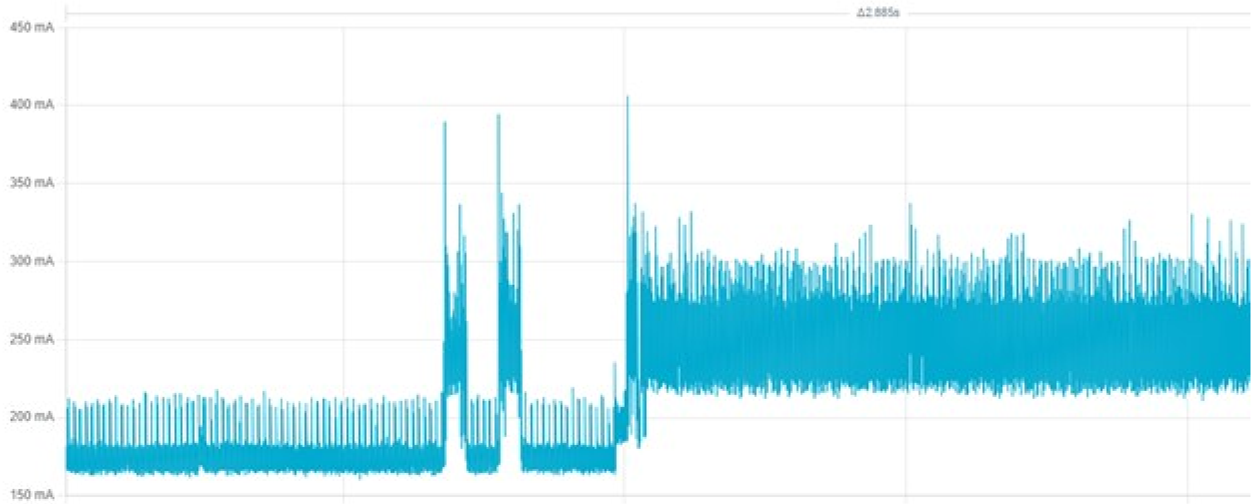

Figure 9. Current consumption when device sends a message

Receiving Data:

Conversely, in this state, the device is configured to receive incoming data. It enables us to discern the power consumption profile during data receiving state. As in the case of sending, the device first establishes an MQTT connection and then receives the message. In addition, a similar jump occurs as in the previous case, with an average current of **243.67 mA**.

In order to compute power consumption metrics in watts, the current measurements must be multiplied by the source voltage, which in our experiment is 3.3V. Therefore, for the average values of power consumption at key moments, we obtain the values listed in Table 4.

Table 4. Power consumption of the device

	Average power consumption
Active State - Communication Module Off	418.64 mW
Active State - Communication Module On	753.23 mW
Communcation module configuration	829.92 mW
Sensor Measurements	635.84 mW
Sending Data	819.85 mW
Receiving Data	804.11 mW

The results from the table provide us with relatively high values for power consumption when we talk about IoT devices. It should be noted that no mechanisms for reducing consumption were used in the implementation. The results presented show that the device consumes the most power in interaction with the communication module. Since it is an LTE module that enables connection to several technologies, it is understandable that the highest power consumption occurs when working with this module. This measurement method can also be applied to other devices and other communication protocols. Similarly, this approach can be applied when measuring the power consumption of devices during the execution of models in federated learning and mutual device communication in an edge environment.

4 Federated learning on heterogeneous nodes with dataset from VisionKG

The general concept of federated learning (FL) involves multiple clients performing machine learning training, each on their own datasets, starting from an initial model received from a central server. Each client, after a predetermined number of epochs, sends the parameters of its updated model to the server, whose tasks is to aggregate, in accordance with a chosen strategy, the collected parameters into a single model, and send it back to the clients, which can then continue their training process. These steps are repeated for several rounds as configured in the server.

For this demonstrator, our goal was to implement federated learning on edge devices using the Flower framework as a federated learning framework providing straightforward structure for exchanging models between clients and server [11], and Vision Knowledge Graph (VisionKG) for querying images from various datasets via their Web API [12].

4.1 Experiment setup

4.1.1 Hardware and prerequisites

Our experimental environment consists of one server and eight edge devices. Demonstrating heterogeneity, local devices participating in training are of three different types as shown in Figure 10: Raspberry Pi 4 (3) [13], NVidia Jetson Nano (4) [14] and NVidia Jetson AGX Orin (1) [15]. All the devices and the server were deployed in the same local network.

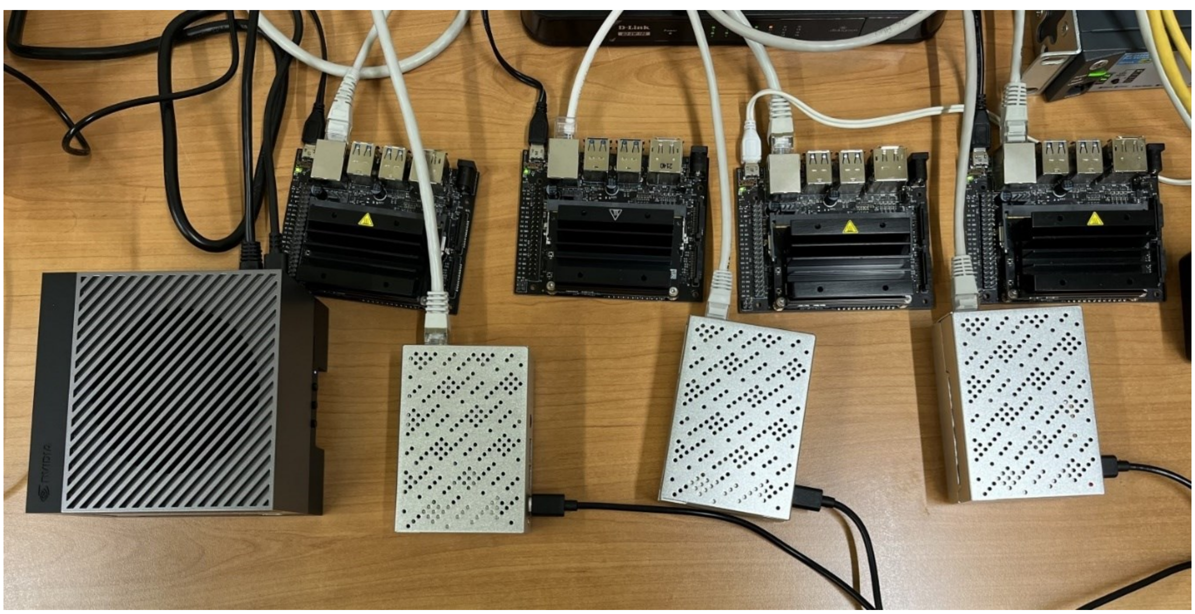

Figure 10. Edge devices used for federated learning experiments.

The Jetson Nano are set up with JetPack version 4.6 [16], the most recent available and compatible version for this device type, running on Ubuntu 18.04.6 LTS OS that is flashed onto a 64GB SD card. Among other packages, this JetPack includes CUDA 10.2.460 version. In order to match up to Flower framework that requires at least Python 3.8, the default Python 3.6.9 version in the JetPack 4.6 is not used, but the virtual environment is set up with Python 3.8 and other suitable packages. The board contains a NVidia Maxwell GPU featuring 128 CUDA cores and quad-core ARM Cortex-A57 CPU running at 1.43 GHz.

The Raspberry Pi 4 are set up to run with Debian GNU/Linux 12 (bookworm) OS on 64GB SD Card. Python 3.11.2 as a default version is compatible with Flower, so it is used to run the client on this device type. RPI4 is characterized by four Cortex A72 CPU cores running at 1.5 GHz.

Similar to Jetson Nano, Jetson AGX Orin is set up with JetPack 5.1.2 version [17], running on Ubuntu 20.04.6 LTS OS, including CUDA 11.4.19 version. It possesses advanced capabilities compared to Jetson Nano in terms of data processing, holding a 1.3GHz NVidia Ampere GPU with 2,048 CUDA cores and 64 Tensor cores, and a 12-core 2.2GHz NVidia Cortex-A78AE ARMv8.2 running at up to 2.2GHz. Default Python 3.8.10 version for the respective JetPack is used for running the client.

In the experiment, we utilized the CPU for Raspberry Pi and Jetson Nano, while Jetson AGX Orin employed its GPU to train the data.

We also attempted to set up a different scenario wherein one Jetson Nano would utilize its GPU for the training. However, we encountered difficulties in configuring this setup with the Flower framework. The appropriate Tensorflow package in regard to this device's architecture (CUDA 10.2) and JetPack version 4.6 specifically requires Python 3.6, which conflicts with the prerequisite of at least Python 3.8 when running a script involving the Flower framework (flwr library).

4.1.2 Software

Code for running the federated learning experiment was implemented using Tensorflow version 2.13.1 and Flower version 1.7.0. Specifically, for the client acting on the Jetson AGX Orin device, Tensorflow 2.12.0+nv23.6 is used [18], so it is compatible with CUDA 11.4.19 version on the device.

Our federated learning experiment involves training a simple convolutional neural network on eight devices, for digit image classification. VisionKG serves as the primary data source for this experiment. VisionKG is a framework developed by researchers at Technical University of Berlin, which enables easy access and querying of state-of-the-art computer vision datasets from diverse sources, used for detection and classification tasks. VisionKG incorporates widely used computer vision datasets, from which annotations and features are extracted using a Visual Extractor, while RDF Mapping Language is utilized to map extracted data into RDF. Data is also linked with various knowledge bases like WordNet and Wikidata. Each label within a specific dataset is implemented through ontology hierarchy utilizing The Semantic Enrichment Reasoner. VisionKG offers a SPARQL web interface for users to access data from interconnected datasets using specific labels, tasks, annotations, etc. Through this interface, data is accessed via SPARQL, allowing users to specify their requirements for data using graph query patterns. Additionally, by using SPARQL, VisionKG API enables users to query desired data from diverse built-in datasets through a few lines of code. Initially, the SPARQL query is constructed within the code, specifying the criteria for the data that will be retrieved from VisionKG. The formulated query is then sent to the VisionKG API to retrieve image data from a semantic knowledge graph. The VisionKG API provides a list of image URLs that can be used to download the images. In our custom implementation, the list undergoes splitting, allocating 80% images for training and 20% for validation of the model. Subsequently, we retrieve all the data from the VisionKG API using the provided URLs. This fetched dataset is then stored locally on devices, making it ready for integration into a machine learning pipeline.

The dataset utilized for our task is an image classification of handwritten digits from 0 to 9, originating from MNIST dataset, queried through VisionKG platform. Each participating client in the federated learning process retains a subset of these images for training purposes. Due to limitations imposed by the SPARQL Engine on VisionKG, a maximum of 400 images per digit are stored on each client. It's worth noting that the data distribution across clients follows the principle of independent and identically distributed (IID) data. This means that each client's dataset is representative of the broader MNIST dataset and is statistically similar in terms of its composition and characteristics. By distributing the data in this manner, we simulate having multiple datasets from multiple sources.

A simple convolutional neural network (CNN) is defined using TensorFlow's Keras API for this image classification task. The CNN architecture consists of 4 convolutional layers (Conv2D), each with a Rectified Linear Unit (ReLU) activation function. Additionally, it includes max-pooling layers (MaxPool2D) and Dropout layers (Dropout). Near the end of model architecture output from the convolutional layers is set to flatten to a 1D array (Flatten), followed by two fully connected (Dense) layers with ReLU activation functions and dropout to further prevent overfitting. The final layer uses a dense layer with softmax activation function for multi-class classification, with the number of units equal to the number of classes, which is 10 in this experiment. The input data consists of grayscale images measuring 28x28 pixels, while the output vector represents the probability of the input image belonging to each class. The class with the highest probability is chosen as the predicted class for a given input.

Given that our classification task involves multiple classes, where the output indicates the probability of the input belonging to each class, we employ one-hot encoding on the true labels. This is achieved by creating a vector with the same number of elements as there are classes and assigning a value of 1 to the element in the vector corresponding to the index of the class, while setting all other elements to 0. The model is compiled using the RMSprop algorithm for optimization, chosen for its ability to achieve more stable convergence and faster learning by dynamically adjusting the learning rates for each parameter based on recent gradients. The selected loss function is categorical cross-entropy, which is typically used in multi-class classification scenarios. It calculates the cross-entropy between the true labels and the predicted probabilities produced by the model. This function is commonly applied when the target variable is categorical and contains more than two categories. The selected evaluation metric for the model is accuracy.

Federated learning involves bidirectional communication, where both the client and the server can send and receive data concurrently. GRPC (gRPC Remote Procedure Call) supports bidirectional streaming, allowing clients and the server to establish long-lived RPC connections for efficient data exchange. Once the server-side script is executed, clients sequentially establish connections with the server. The federated learning process initiates as multiple client-side scripts run, adhering to the minimum required number of clients for training, as defined within the strategy. During the initiation of federated learning, the parameters of a global model are initialized and distributed to each client, marking the beginning of the federated learning process. In each iteration of federated learning, every client executes the fit method, where it sets the weights of its local model to the parameters received from the server, trains the model using its local training dataset for one epoch, and then transmits the updated parameters back to the server. Each client also evaluates the performance of its model on its local validation dataset. Then, the server combines all received weights from each client using a specified Strategy to compute the parameters of the global model. The learning process on the server side can be customized through the Strategy abstraction in Flower. We used a built-in strategy FedAvg (Federated Averaging) where the weighted average of received model weights from the clients is computed, with each set of weights weighted by the number of examples used by the respective client during training. The parameters defined through the strategy concern prerequisites for the number of clients participating in each training round, ensuring efficient coordination and utilization of distributed resources. The iterative process of exchanging weights between clients and the server continues for the specified number of rounds. For our setup, we configured the model to undergo ten rounds of training across all eight available clients.

4.2 Results

In the initial experiment, each of the 8 clients uses one-eighth of the dataset obtained from VisionKG, where 20% of data is used for validation. The federated learning process lasts for 10 rounds. In Figure 11, we observe the final round of the federated learning process across all devices. The iotlab@rpi4-x terminals execute client processes on Raspberry Pi 4 devices, iotlab@jetnano-x terminals run client processes on Nvidia Jetson Nano, and iotlab@agxorin-1 represents a terminal running a client process on NVidia Jetson AGX Orin. The terminal labelled iotlab@flserver-1 corresponds to the terminal executing server code on the server computer. In the final round, every client achieves a validation accuracy of 90% or higher, with the client operating on NVidia Jetson AGX Orin attaining the lowest accuracy of 90% and the client running on one of the Raspberry Pi 4 devices achieves the highest accuracy of 96.3%.

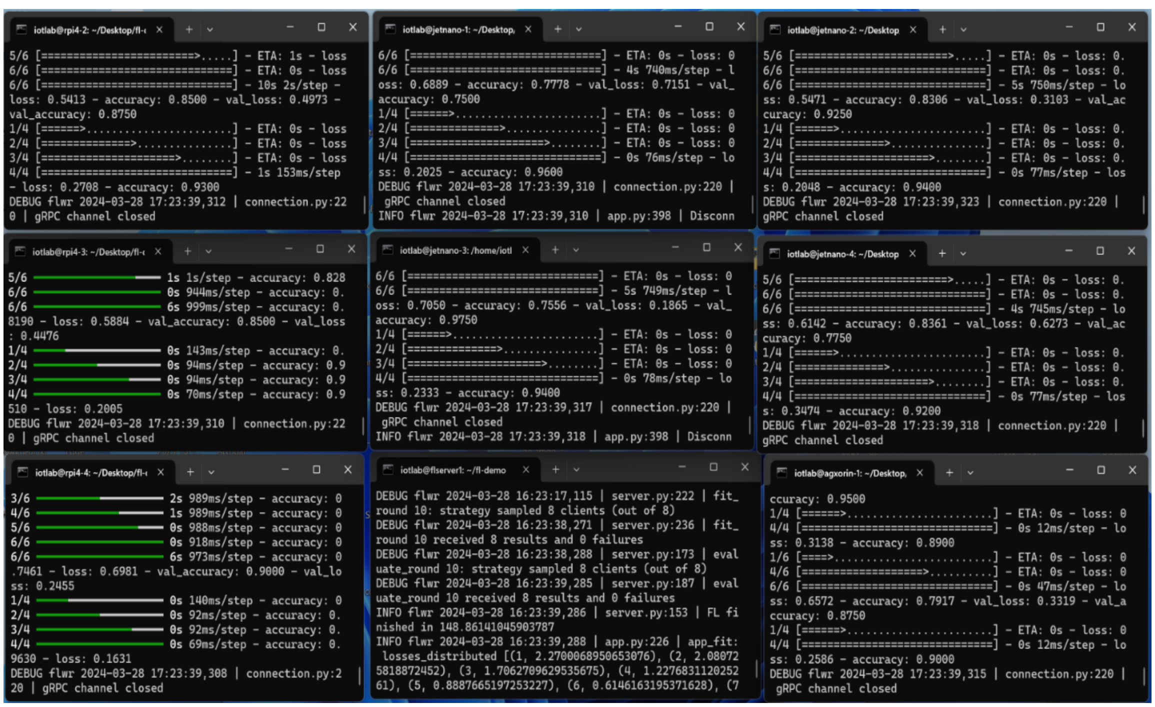

Figure 11. Last round of federated learning experiment on eight different devices

TensorBoard is utilized to visualize the average metrics during the training and testing of local models on devices. This process involves transmitting accuracy and loss information from clients to the server at the end of each round and writing this data to log files, which TensorBoard then utilizes to generate graphs. Figure 12 illustrates the average accuracy and loss during the training and testing of local models after each round across all clients.

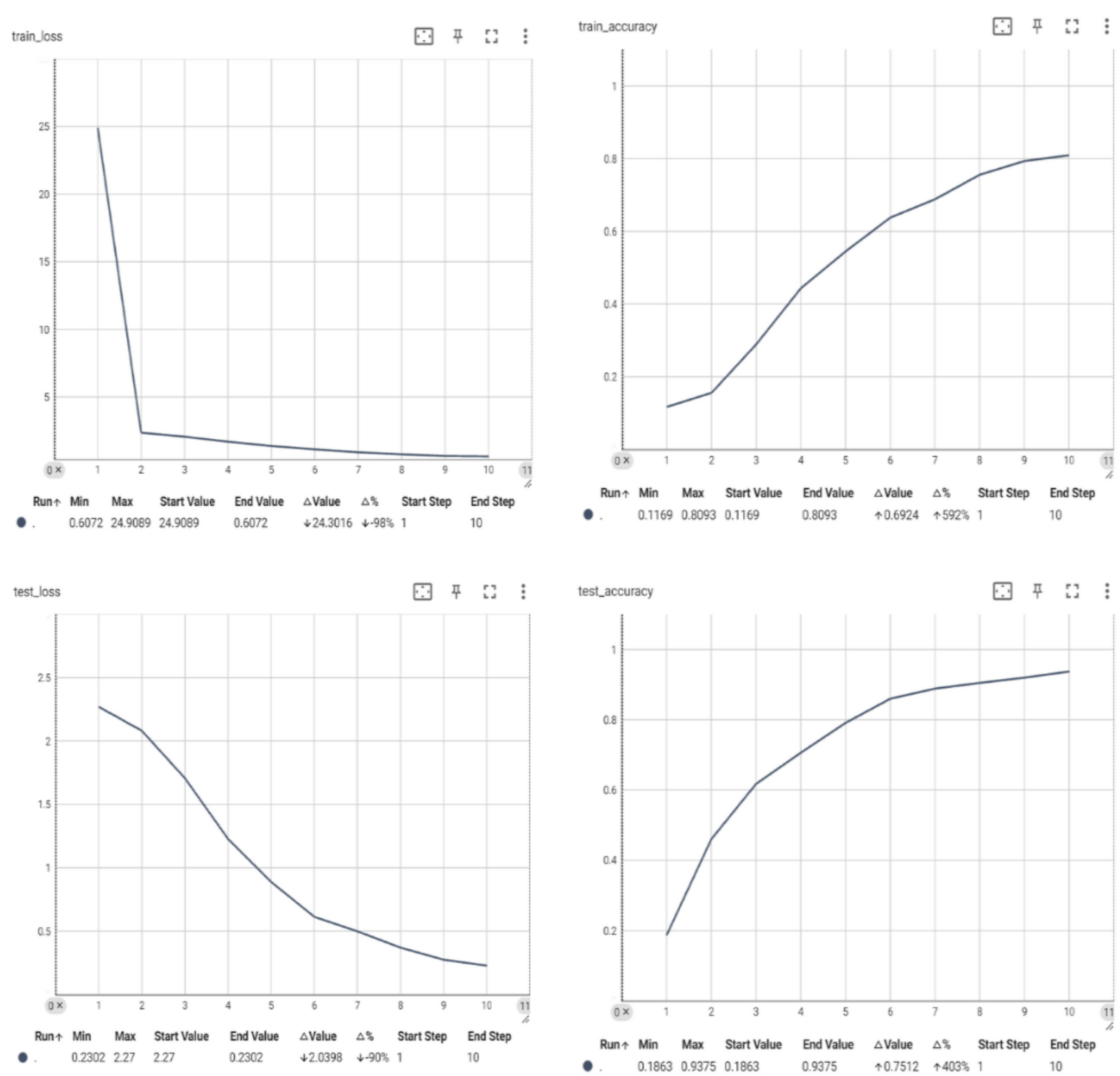

Figure 12. The mean loss and accuracy averaged across all local models over multiple rounds, derived from their respective training and testing datasets

Once a round of federated learning finishes, the client and server communicate by exchanging parameters of the client's local and the server's global model. To monitor the traffic on the port where client and server communicate, 'tcpdump' is employed. Upon completing one round of learning, the client transmits its local model parameters to the server, utilizing approximately 1340 HTTPS packages that carry around 1.94 MB of data. Next, approximately 234 HTTPS packages are sent from the server to the client, containing the parameters of the global model after averaging weights from clients. These packages contain approximately 1.95 MB of data.

Another experiment was conducted to demonstrate the variance in learning speed between different hardware configurations between devices. Specifically, the experiment compared the training performance of one Raspberry Pi 4 device utilizing CPU, one NVidia Jetson Nano device training the model on CPU, and NVidia Jetson AGX Orin device employing GPU for training. This time, the experiment utilized the MNIST dataset directly downloaded from the Keras API in the code. By downloading 60 thousand images, which were then split among the devices, the dataset size increased, consequently extending the time needed to complete a single epoch of training. The prolonged duration for completing a single epoch highlighted the differences in training speed across the various hardware setups between devices. Figure 13 illustrates the completion of the first round in the experiment. The Raspberry Pi 4 device required approximately 208 seconds to finish the first round of training, while the NVidia Jetson Nano needed about 174 seconds, and the NVidia Jetson AGX Orin took approximately 23 seconds. These results indicate that training on the NVidia Jetson AGX Orin using GPU is approximately 7 times faster than on the CPU of the NVidia Jetson Nano, and about 9 times faster than training on the CPU of the Raspberry Pi 4.

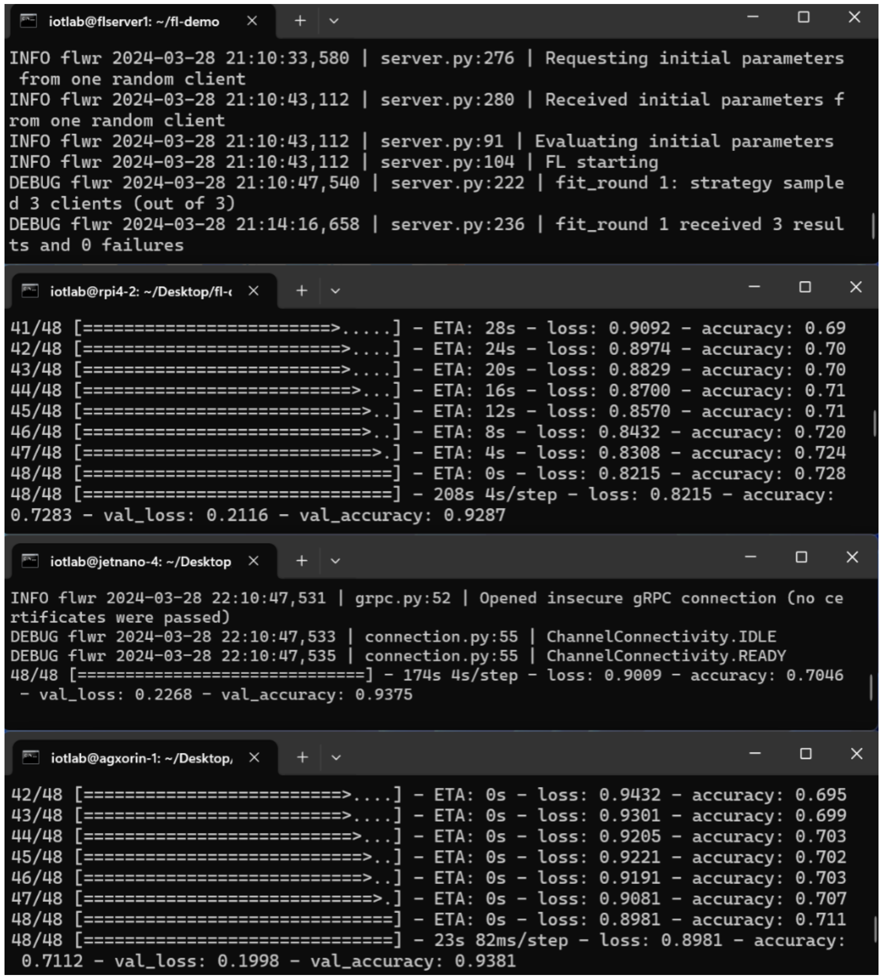

Figure 13. Completion of the first round of the experiment

5 Conclusion

This deliverable presents a demonstrator of state-of-the-art mechanisms for edge orchestration, energy efficiency, and ML workflows through live experiments conducted in the UNIZG-FER testbed. The goal of this demonstrator was to explore and assess mechanisms suitable for integration into a data-driven orchestration middleware for energy-efficient IoT systems supporting ML workflows.

The demonstrator unfolded in three parts. In the first part, we focused on service orchestration at the edge and conducted two experiments. Firstly, we evaluated the performance of container orchestration tools at the edge, revealing Kubernetes and its edge-aware distributions, such as K3s, as optimal options for service orchestration. Subsequently, we conducted an experiment demonstrating the execution of edge orchestration workloads in an emulated network using K3s on the Imunes tool, showcasing the ease of validating edge orchestration solutions in conditions closely replicating actual deployment conditions.

The second part concentrated on measuring power consumption of a resource-constrained IoT device, aiming to identify the primary power consumers relative to the total device consumption for a given IoT use case. The experiment, conducted on an STM32 device, revealed that the highest power consumption occurred during interactions with the communication module, particularly the LTE module enabling connections to multiple technologies. The results underscored the importance of implementing mechanisms to reduce power consumption due to the high energy consumption observed in various states.

Lastly, the third part demonstrated a federated learning deployment on heterogeneous nodes within the UNIZG-FER testbed. The experiment, ran with a dataset from VisionKG developed by TUB on a testbed comprising eight edge devices (three Raspberry Pi's, four Nvidia Jetson Nano's, and one Nvidia Jetson AGX Orin), showcased the complexity of deploying federated learning on diverse hardware. Two experiments were conducted to assess network traffic and training time across different hardware. Results highlighted the advantages of utilizing specialized GPU hardware for ML tasks on Nvidia Jetson AGX Orin, significantly reducing training time compared to CPU-based systems such as Nvidia Jetson Nano or Raspberry Pi 4. The network overhead for uploading a local model and downloading an aggregated model from the server was approximately 1.95 MB, indicating relatively low traffic. However, in scenarios with thousands of nodes, the central server could face overload. Therefore, investigating hierarchical federated learning configurations is an important next step to mitigate potential scalability issues.

6 Acronyms

AI	Artificial Intelligence
AIoT	Artificial Intelligence of Things
D&C	Dissemination and Communication
ECC	Edge-to-Cloud Continuum
FL	Federated Learning
IoT	Internet of Things
MQTT	Message Queuing Telemetry Transport
PPK	Power Profiler Kit
QoS	Quality of Service

7 List of Figures

8 List of Tables

9 References

[1] Čilić, I., Podnar Žarko, I., Kušek, M.: Towards service orchestration for the cloud-to-thing continuum. In: 2021 6th International Conference on Smart and Sustainable Technologies (SpliTech). pp. 01–07 (2021)

[2] Čilić, I.; Krivić, P.; Podnar Žarko, I.; Kušek, M. Performance Evaluation of Container Orchestration Tools in Edge Computing Environments. Sensors 2023, 23, 4008.

[3] University of Zagreb: Imunes, http://imunes.net

[4] Mikroelektronika, https://www.mikroe.com/

[5] Uni Clicker, https://www.mikroe.com/uni-clicker

[6] Environment click, https://www.mikroe.com/environment-click

[7] LTE IOT 9 click, https://www.mikroe.com/lte-iot-9-click

[8] Power Profiler Kit II, https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2

[9] MQTT, https://mqtt.org/

[10] AT Commands, https://download.mikroe.com/documents/datasheets/EXS62-W%20AT%20Commands.pdf

[11] Flower framework, https://flower.ai/

[12] VisionKG, https://vision.semkg.org/explorer.html

[13] Raspberry Pi 4 Tech Specs, https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[14] Jetson Nano, https://developer.nvidia.com/embedded/jetson-nano

[15] Jetson AGX Orin, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/

[16] JetPack SDK 4.6, https://developer.nvidia.com/embedded/jetpack-sdk-46

[17] JetPack SDK 5.1.2, https://developer.nvidia.com/embedded/jetpack-sdk-512

[18] Tensorflow for 5.1.2 JetPack, https://developer.download.nvidia.com/compute/redist/jp/v512/tensorflow/

