A Secure and Reusable Artificial Intelligence Platform for Edge Computing in Beyond 5G Networks

D4.2. Results on the validation of the AI@EDGE Connect-Compute Platform

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101015922

D4.2. Results on the validation of the AI@EDGE Connect-Compute Platform	
WP	WP4 – AI@EDGE Connect-Compute Platform
Responsible partner	ICCS
Version	2.4
Editor(s)	George Lentaris (ICCS)
Author(s)	George Lentaris, Achilleas Tzenetopoulos, Dimitris Danopoulos, Aimilios Leftheriotis (ICCS), Estefanía Coronado, Javier Palomares, Miguel Catalan-Cid, Jorge Pueyo (I2CAT), Cristina Costa, Arfan Haider Wahla Raman Kazhamiakin, Matteo Martini (FBK), Chi-Dung Phung, Mario Patetta, Stefano Secci (CNAM), Daniele Ronzani (HPE), Babak Mafakheri (SPI), Shuai Zhu (RISE), Antonino Albanese (ITL), Javier Renart, Enrique Lluesma (ATOS), Omar Anser (Inria)
Reviewer(s)	Nicola di Pietro (HPE), Babak Mafakheri (SPI), Roberto Riggio (UNIVPM), Bengt Ahlgren (RISE)
Deliverable Type	R
Dissemination Level	PU
Due date of delivery	2023-09-30
Submission date	2023-10-03

Version History

Version	Date	Author	Partner	Description
0.1	19/5/2022	George Lentaris Cristina Costa	ICCS FBK	ToC
0.2	21/06/2022	Estefanía Coronado Javier Palomares	i2CAT	Section 2.6 (Integration of CCP)
0.3	28/06/2022	ALL	ALL	Sections 2-4, all subsections final contributions
0.4	30/06/2022	George Lentaris	ICCS	Executive summary, Introduction, Conclusions

0.4	4/06/2022	George Lentaris	ICCS	Final formatting
0.5	6/7/2022	Nicola di Pietro, Babak Mafakheri	HPE, SPI	Reviewed document
0.6	7/7/2022	George Lentaris, ALL	ALL	Contributions from addressing final review
1.0	8/7/2022	Irene Facchin	FBK	Final Review of draft 1
1.1	26/11/2022	George Lentaris, Raman Kazhamiakin	ICCS FBK	Started work on draft 2
1.2	26/01/2023	ALL	ALL	Completed new contributions
1.3	10/2/2023	Nicola di Pietro, Babak Mafakheri	HPE, SPI	Reviewed document
1.4	10/2/2023	George Lentaris	ICCS	Format & Close 2nd draft
2.0	13/2/2023	George Lentaris, Raman Kazhamiakin	ICCS FBK	Started final D4.2
2.1	1/9/2023	ALL	ALL	Included new inputs, including results (sec. 4)
2.2	21/9/2023	Nicola di Pietro, Bengt Ahlgren, Roberto Riggio	HPE, RISE UNIVPM	Document review
2.3	28/9/2023	ALL	ALL	Document corrections
2.4	03/10/2023	Irene Facchin	FBK	Final D4.2 submitted

Table of Contents

List of Figures

List of Tables

Glossary	
3GPP	3rd Generation Partnership Project
4G	4th Generation of mobile communication networks
5G	5th Generation of mobile communication networks
5GAA	5G Automotive Association
5GC	5G Core network
6G	6th Generation of mobile communication networks
AGV	Automated Guided Vehicle
AI	Artificial Intelligence
AIF	Artificial Intelligence Function
AMF	Access and mobility Management Function
APN	Access Point Name
AR	Augmented Reality
AUSF	Authentication Server Function
BVLOS	Beyond Visual Line of Sight
CCP	Connect Compute Platform
C-V2X	Cellular Vehicular communication
COTS	Commercial Off-The-Shelf
CP	Control Plane
CPU	Central Processing Unit
CU	Centralized Unit
DE	Deliverable Editor
DL	Downlink
DNN	Data Network Name
DNS	Domain Name System

DSP	Digital Signal Processing
DU	Distributed Unit
EDA	Electronic Design Automation
EPC	Evolved Packet Core
FaaS	Function as a Service
FL	Federated Learning
FPGA	Field-Programmable Gate Array
FPV	First Person View
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GPU	Graphic Processing Unit
HIL	Hardware In the Loop
HITL	Human-in-the-loop
HLS	High Level Synthesis
HSS	Home Subscriber Server
HO	HandOver
HW	HardWare
ICT	Information and Communication Technology
ICS	Information Coordination System
IFE	In-Flight Entertainment
IIoT	Industrial Internet of Things
IOC	the Intelligent Orchestrator Component
IoT	Internet of Things
IoU	Intersection over Union
IP	Internet Protocol

I-UPF	Intermediate User Plane Function
KPI	Key Performance Indicator
LCM	Lifecycle Management
LSTM	Long Short Term Memory
LUT	Look Up Table
mAP	Mean Average Precision
MEC	Multi-access Edge Computing
MEO	MEC Orchestrator
ML	Machine Learning
MLP	Multi Layer Perceptron
MME	Mobility Management Entity
mMTC	massive Machine-Type Communication
MNO	Mobile Network Operator
MQTT	Message Queuing Telemetry Transport
MTC	Machine Type Communications
MTO	Multi Tier Orchestrator
MPTCP	Multipath Transmission Control Protocol
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NFVO	Network Function Virtualization Orchestrator
NRF	Network function Repository Function
NSA	Non-StandAlone
NSSAI	Network Slice Selection Assistance Information
NSSF	Network Slice Selection Function
OWL	Web Ontology Language

PCF	Policy Control Function
PCIe	Peripheral Component Interconnect express
PCRF	Policy and Charging Rules Function
PDCP	Packet Data Convergence Protocol
PDU	Protocol Data Unit
PGW	Packet data network GateWay
QoS	Quality of Service
R16	Release 16
RAN	Radio Access Network
RAT	Radio Access Technology
REST	REpresentational State Transfer
RIC	RAN Intelligent Controller
RNIS	Radio Network Information Service
ROS	Robot Operating System
RSRP	Reference Signal Received Power
RSRQ	Reference Signal Received Quality
RSU	Road Side Units
RTL	Return To Launch
RU	Radio Unit
SA	StandAlone
SBA	Service-Based Architecture
SDN	Software-Defined Networking
SGW	Service GateWay
SMF	Session Management Function
S-NSSAI	Single – Network Slice Selection Assistance Information

TAC	Tracking Area Code
TRL	Technology Readiness Level
UC	Use Case
UE	User Equipment
UL	Uplink
UP	User Plane
UPF	User Plane Function
URLLC	Ultra-reliable Low Latency Communications
VNF	Virtual Network Function
VR	Virtual Reality
V2I	Vehicle to Infrastructure
V2N	Vehicle to Network
V2V	Vehicle to Vehicle

Executive Summary

This document constitutes a final report on the results of the implementation and validation of the AI@EGE Connect-Compute Platform. The achievements reported here show the progress towards fulfilling the project's overall Objective 4 "To design, prototype, and validate a Connect-Compute Platform supporting perceived zero-latency services using a mix of computing and connectivity resources". The report summarizes the work carried out in the scope of "WP4. Connect-Compute Platform" in the final period of AI@EDGE. Its main purpose is to describe the testbed & methodologies used for validating the final Connect-Compute Platform, along with the evaluation results.

The contents of D4.2 rely on and reference the contents of D4.1 "Design and initial prototype of the AI@EDGE Connect-Compute Platform". More specifically, D4.2 focuses on extending and evaluating the developments of D4.1 regarding the Connect-Compute Platform design; the HW accelerated solutions for AI/ML; the data-driven service lifecycle management solutions for the deployment, management, and monitoring of end-to-end AI-enabled applications supported by the Serverless solutions; cross layer, multi-connectivity-enabled disaggregated RAN; integration of the Connect-Compute Platform components and the orchestration interfaces. Overall, main research objectives underlying this report are: i) design and validate a Connect-Compute Platform ii) extend ETSI MEC/NFV architectures with applications and models capable of providing the AI@EDGE platform with the context and metadata necessary to take automatically actionable decisions and to realize intelligent data and computation offload control and management of applications and services deployed over the decentralized and distributed AI@EDGE platform, iii) investigate different hardware acceleration solutions (FPGA, GPU, CPU) spanning from the terminals to the cloud for highly decentralized and distributed workload management, iv) analyze and compare dual-connectivity monolithic RANs with cross-layer multi-connectivity disaggregated RANs to see if the network topology should be dynamically adapted to evolving network conditions.

Given the above scope, D4.2 consists of two main parts: 1) the report of design choices and qualitative review of the HW & SW developments in the Connect-Compute Platform (components and aspects developed/extended after D4.1), and 2) the quantitative evaluation of all the components and the integrated Connect-Compute Platform, based on the testbeds & methodologies followed by actual experimental results. The former part is summarized in Section 2, whereas the latter part is given in Sections 3 and 4.

Overall, the work performed on the Connect-Compute platform and the evaluation results demonstrate considerable robustness and efficiency in the AIF orchestration and life-cycle management tasks across different layers of the architecture. The use of AI-specific solutions, involving Serverless platforms, demonstrated its advantages for the zero-touch automation of the life-cycle management tasks, such as monitoring, autoconfiguration, and continuous ML model updates. Furthermore, effective resource management of the acceleration resources for the purpose of the AIF orchestration allows for achieving an order of magnitude improvement in the compute capabilities of the Connect-Compute Platform. Similarly, the proposed approaches for the non-RT intelligent control-loops as well as MPTCP proxy and predictive scheduling solutions demonstrate significant improvements regarding the static resource allocations and traffic management.

1 Introduction

One of the main goals in AI@EDGE is to build a Connect-Compute Platform (CCP) and showcase secure and automated management, orchestration, and operation of AI-powered services over edge and cloud compute infrastructures, with close to zero-touch of the underlying heterogeneous MEC resources (network, storage, and compute resources), as well as acceleration of AI functions towards improved computational performance across the edge-cloud continuum. To achieve such goals, in WP4 we develop and integrate solutions managing distributed resources inside the infrastructure, with the support of a variety of virtualization technologies (VMs, containers, K8s pods, serverless platforms), which enable a very fine-grained exploitation of the available resources. The platform supports a distributed, multi-layer cloud deployment where orchestration mechanisms take place within both the cloud domain (centralized) and the edge domain (distributed). Furthermore, it will leverage heterogeneous hardware acceleration solutions (CPU, GPU, FPGA) to optimize various metrics (energy consumption, performance, security) for specific AI-based workload types, and it will also leverage disaggregated 5G Radio Access Network (RAN) supporting beyond R16 cross-layer multi-connectivity.

The Connect-Compute Platform is a one of the core elements of AI@EDGE. This deliverable is linked with the technical work carried out across WP2, WP3, and with the validation scenarios described in WP5. Therefore, to understand the content of this report, we suggest the following reading path:

1. D2.1 *"Use cases, requirements, and preliminary system architecture"*
2. D2.2 *"Prelim. assessment of system architecture, I/F specifications, technoeconomic analysis"*
3. D3.1 *"Initial report on systems and methods for AI@EDGE platform automation"*
4. D3.2 *"Final report on systems and methods for AI@EDGE platform automation"*
5. Most importantly: D4.1 "*Design & initial prototype of AI@EDGE Connect-Compute Platform*"

In particular, the current report includes the results of the validation of the AI@EDGE Connect-Compute Platform developed in WP4. The validation is performed both in terms of a high-level review of design choices and component developments, as well as in terms of a systematic evaluation of the platform's performance. For a more comprehensive performance evaluation, the latter is done both at component/feature level and at system integration level.

Structurally, this report is divided into two main parts, one qualitatively describing the development results (e.g., final design choices, successful technical solutions, development steps with examples) and one quantitatively reporting the performance results. The former is given in Section 2 "*Architectural Components of the CCP: design & development update*", while the latter is given in Section 3 "*Validation Methodology of the CCP*" plus Section 4 "*Experimental Results*", i.e., it is further subdivided into the methodologies/testbeds devised for validating the platform and the actual results measured when following these methodologies. We note that the first part (Section 2) describes the implemented architecture/components of the Connect-Compute Platform developed in WP4, i.e., it describes a specific instantiation of the more general architecture presented in WP2. Furthermore, we note that the descriptions in Section 2 are strongly related to the contents of D4.1, and thus, to avoid repetitions with D4.1, Section 2

should be considered as an extension reporting only the relevant updates and making numerous references to D4.1.

In a more detailed outline, Section 2 is divided into subsections following the structure of D4.1 and analysing the new developments per platform component/feature. For brevity, it begins with a succinct description of the updates versus D4.1, i.e., what is the new development (the "delta" vs "D4.1"), and further analysed in the D4.2 subsections. Section 3 presents the basic integration testbed of AI@EDGE (FBK premises) facilitating the evaluation of the Connect-Compute Platform, together with an auxiliary testbed (ICCS premises) facilitating more in-depth experiments for MEC acceleration. Section 3 also presents the methodologies to be used when evaluating the performance of the Connect-Compute Platform: it is further subdivided in subsections, one per platform/feature that we validate, which present customized "scenarios" with specific goals and steps to follow during testing. Each scenario will be executed upon completion of the Connect-Compute Platform to gather performance metrics per component/feature, as well as for the entire integrated Connect-Compute Platform. The results are presented and assessed accordingly in Section 4. Finally, Section 5 concludes the document and describes future steps.

2 Architectural Components of the CCP: design & development update

This section serves the purpose of summarizing and explaining certain design choices and developments in the AI@EDGE Connect-Compute Platform and in essence, it forms the extension of D4.1 [1] . To avoid repetitions, the current section presents only the changes/updates with respect to D4.1. For brevity, we give a succinct description of the CCP below, followed by a subsection summarizing the updates compared to D4.1 (the new developments), and multiple subsections describing the details of each development, per component/feature of CCP.

The AI@EDGE architecture presented in Figure 1 depicts an updated view of the system architecture presented in D2.2 and D4.1. The architecture is functionally divided into two main blocks, comprising the Network and Service Automation Platform (NSAP), which implements network system automation functions, and the CCP on the bottom, dedicated to handle the infrastructure, network and automation resources to properly manage the AIF orchestration.

Figure 1 depicts the Connect Compute Platform in detail, with the platform components distributed between the Central Office and the Radio Access Sites. In the figure multiple MEC systems are considered, where each MEC system represents a collection of MEC hosts and MEC management processes necessary to run MEC applications. Each MEC host contains a MEC platform and the corresponding virtualization infrastructure, which provides compute, storage, and network resources to MEC applications. The MEC Platform can provide itself with several MEC services, such as the RNIS service. The MEC host is strategically placed by the edges of the network to provide computation and storage capabilities near the access network and provides, among other advantages, lower latency. To this aim, the 5G traffic is steered towards the MEC host where it can be processed.

The deployment and management of applications follows a combination of hierarchical and distributed approaches. The initial request for deployment is meant to be received by NSAP, and specifically to be managed by the multi-tier orchestrator (MTO). MTO decides between Cloud or a particular MEC relying on Intelligent Orchestrator Component (IOC) for this purpose. The deployment is then delegated to the corresponding MEC Orchestrator (MEO). This vision intends that the platform and the applications deployed can continue their execution even if the MTO goes down, since the rest of functionalities would be managed locally at each MEC system. As shown in D4.1, MEO is responsible for the following functions: maintaining an overall view of the MEC system based on deployed MEC hosts, and available resources (in particular, acceleration resources), keeping a record of on-boarded packages; selecting appropriate MEC host(s) for application instantiation based on constraints, such as latency, available resources, and available services; triggering application instantiation and termination; triggering application relocation as needed when supported. The MEC Platform has the functionalities required to run MEC applications on a particular virtualization infrastructure and enables them to provide and consume MEC services. In the project, the role of VIM is taken by Kubernetes, while the one of MEC Platform is taken by the LightEdge software, a microservice-based implementation of the ETSI MEC Architecture.

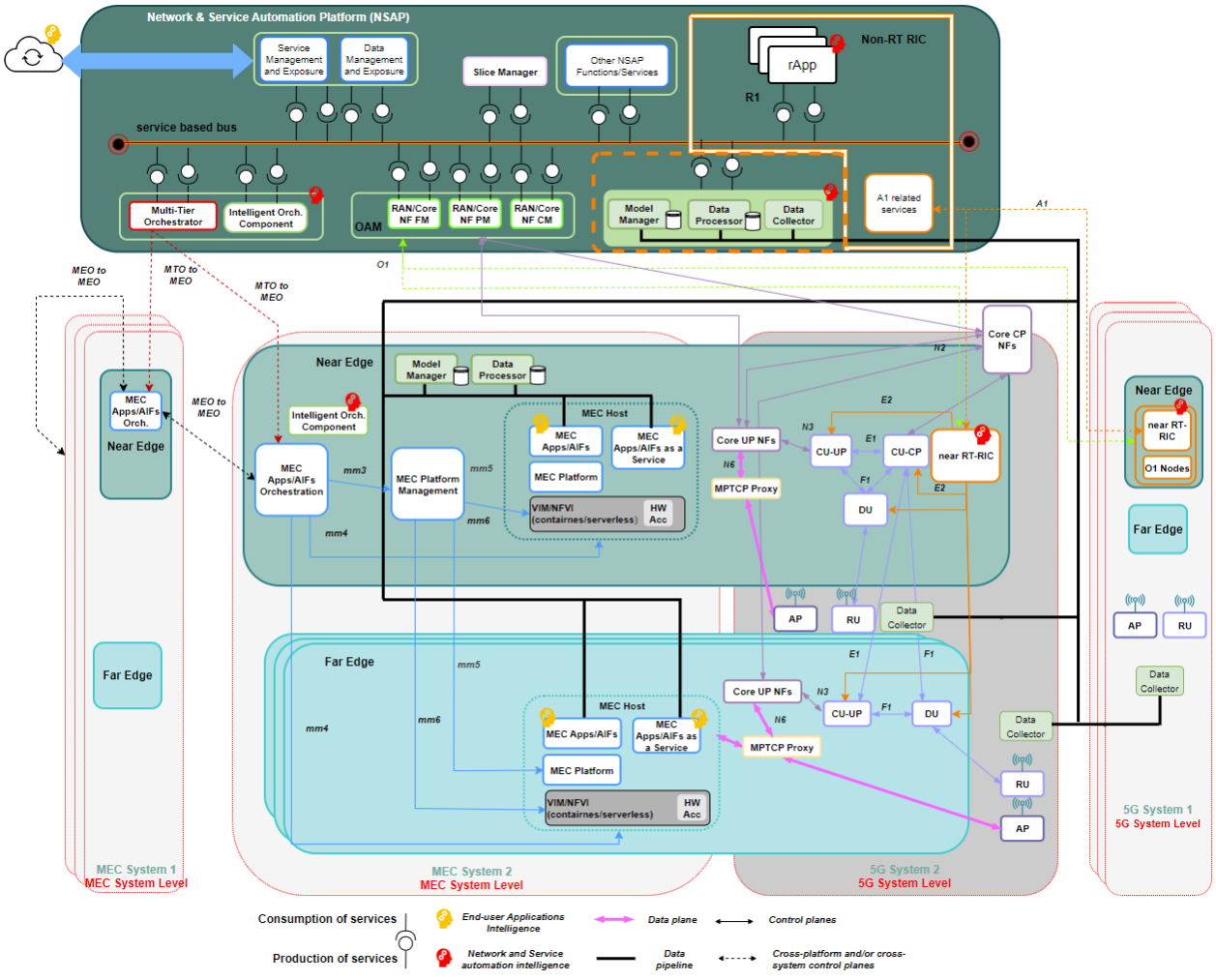


Figure 1 Updated architectural view of the AI@EDGE Connect-Compute Platform

From the grounds and options presented in D4.1 [1], this deliverable takes a clear path regarding the deployment options of the CCP in relation to the ETSI MEC [3] and ETSI MEC in NFV [5] architectures. However, in both cases the functionalities provided by the NSAP and the CCP are the same, and just differ on the interconnection of the modules and how the lifecycle of the applications is handled and developed. For this reason, the following subsections describe the various components involved in the AI@EDGE platform, detail the evolution and actual deployment decisions made within the consortium, and finally explains the integration of the various modules described before.

2.1 Overview of updates compared to D4.1

With respect to the state of the CCP described in D4.1, the following additional work was conducted here towards improving and/or implementing the proposed solutions of AI@EDGE. In the remainder of section 2, each topic is further analysed in its own subsection.

Serverless Platform

1. Integration relying on the LightEdge Runtime with managers, Nuclio and Seldon Core (Section 2.2).
2. Definition and integration of the AI-related LCM functionality into the CCP using Serverless technologies (Section 2.4).

Disaggregated RAN

1. Refinement of the RAN functionalities, interfaces, and integration (Section 2.3).

Data-driven Service Lifecycle for AI-enabled Applications

1. Solutions support the distributed orchestration and lifecycle management phases: monitoring. autoconfiguration, autonomous AI model update (Section 2.4).
2. Final specification of the AIF Descriptor information model with the focus on the AI-related and data-driven aspects of the lifecycle management specification (Section 2.5).

Cross-layer, Multi-Connectivity Aggregation and Scheduling Technologies (Section 2.6)

1. Integration of the MPTCP proxy (off-path mode) in the WP4 testbed.
2. Integration of MPTCP capabilities in UC4 devices and servers.
3. Setup of an emulation testbed for the creation of useful datasets for the evaluation of the algorithms to be proposed in the frame of T4.3.

Hardware Acceleration Solutions for AI/ML (Section 2.7)

1. Actual integration of multiple HW devices over Virtual Machines, using KVM as the hypervisor, Ubuntu as the Operating System, and a single-master Kubernetes (v1.20.5) cluster for the inter-VM orchestration of containers, altogether forming a hypothetical MEC cluster of near-edge and far-edge nodes (with a 1Gbps/10msec Eth link emulating their in-between connection).
2. Development of exemplary accelerated AIFs with Xilinx Vivado HLS, Vitis-AI, and NVIDIA TRT.
3. Proof-of-concept design of an Intelligent Acceleration Resources Manager (IARM) module to support the orchestration of the CCP.
4. Setup of the NetFPGA programming environment with the design of an ad-hoc South-Bound Interface.

Integration of the CCP

1. Consolidation of the architecture of the orchestration subsystem (Section 2.8)
2. Refinement and consolidation of the MTO-MEO-MECPM interfaces (Section 2.9).

2.2 Serverless Platform

As described in the Deliverable D4.1, the integration of Serverless platform functionality in the CCP allows for providing MEC applications with the additional capabilities achieved by the stateless, event-driven functions that can be easily onboarded, scaled, and exploited for various generic purposes making it possible considering the Serverless functions as MEC Apps. More specifically, Nuclio Serverless platform has been evaluated and proposed as a potential implementation. Its integration in CCP is presented in Section 2.2.2.

Apart from various generic scenarios for the Serverless functions usage, in case of an AI-oriented platform like AI@EDGE's CCP, it is important to consider the scenarios related to the AI-specific functionality, such as AI model serving and inference pipelines, model monitoring, in order to support zero touch automation loops that include model re-training in case of deviations (e.g., concept drift), model update, re-deployment, etc. In this document, we consider the way the CCP integrates the Serverless platform and discuss the specific platform extensions that target the AI-specific supporting functionality based on Serverless solutions.

2.2.1 AI-Specific Serverless Functionality

Typically, AI functions expose an inference / prediction / classification functionality that relies on a pre-trained ML model (i.e., model serving scenario). In such a case, the AI function is delivered as a microservice with a standardized API (e.g., REST or gRPC), which accepts the input data, calls the model for the inference, and returns the result. Often such a service can be provided in a standard way and even with the standard interface. With this scenario in mind, the CCP offers a wide range of capabilities supporting the life-cycle of such functions and enabling a zero-touch automation approach for their management. In particular, the AI@EDGE platform supports:

- The automated deployment of AI/ML models as AI functions, via generation of the corresponding AIF descriptors and exploiting pre-configured model server deployment configurations.
- The traceability of the AI function executions in a standard manner, collecting the execution metrics (e.g., model and server latency) and the domain data used for the invocation.
- The automation of the AI monitoring to detect the model confidence degrade (e.g., due to concept drift) through the data collection and continuous deviation analysis.
- Model update via reporting the deviations to trigger the model re-training (if available) and re-deployment of the new AIF versions based on the new models.

The implementation of this functionality fits in the AI@EDGE platform functionality as the supporting mechanisms for the model management described in project Deliverable D3.2 [3].

Serverless Model Serving

For what concerns the exposing the AI models as functions using Serverless frameworks, there is a wide range of solutions already available. Being a typical scenario for the MLOps workflows, also some

standardization efforts have recently emerged. In particular, a joint effort of the Seldon[1], KServe[2], and Nvidia Triton[3] projects led to the definition and implementation of the V2[4] inference protocol, where the interaction with the served model is defined with:

- Inference endpoint
- Health check endpoints
- Metadata points

This protocol allows for the standardized implementation of these wrappers, referred to as model servers. that are then delivered directly by the corresponding Serverless platform. That is, to deploy a new model server, it is necessary to provide the reference to the corresponding standard wrapper (e.g., Docker image, Kubernetes Custom Resource Definition, etc) and the reference to the model file. Depending on the specific framework this may be achieved, for example, as follows:

- In case of Seldon Core framework, the deployment is performed via the corresponding Kubernetes Operator and can be easily automated within the AI@EDGE CCP as the deployment can be defined with the standard Helm chart specification.
- In case of Nuclio Serverless platform, this is implemented as a part of a larger platform MLRun[5], where deployment workflow is further automated. This integration may be brought to the CCP integration in a similar way.

Once deployed, the model server APIs are exposed within the platform and can accept the inference requests.

It is important to note that in case of standard inference model servers, the cold start time is drastically reduced: the reusable server images do not require rebuild and depend only on the model data import. Placing the model registry close to the execution infrastructure and using the appropriate caching solutions may further improve this aspect.

Serverless Model Tracing

Another important functionality provided by the model serving scenario is the automation and standardization of the model execution tracing. More specifically, given the relevance of the AIF performance for the orchestration, it is necessary to collect e.g., the information about the execution latency

[1] https://docs.seldon.io/projects/seldon-core/en/latest/index.html#
[2] https://kserve.github.io/website/0.9/
[3] https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html
[4] https://docs.seldon.io/projects/seldon-core/en/latest/reference/apis/v2-protocol.html
[5] https://www.mlrun.org/

of the inference model. Similarly, in order to make decisions about the model quality and possible concept drift, it is necessary to continuously observe, collect, and evaluate the processed data.

Having the standard model server allows for collecting such information in a standard manner, without implementing ad-hoc functionality within the function itself. Figure 2 illustrate the model monitoring approach engaged by the MLRun framework: the metrics and data are collected and passed to a model monitoring function on top of the collected data. The results of the analysis may be further exposed as alerts and visualized to the operator.

To enable such an approach, the underlying model servers should provide the data collection functionality out of the box. Depending on the specific implementation, this may be achieved, e.g., as follows:

- In case of Seldon Core framework it is possible to enable the standard metrics regarding the request load, latency, etc. The metrics are collected by Prometheus. It is also possible to define and expose custom metrics in addition to the standard ones. As for the data collection, framework allows for passing the input and output data to some collector, such as Apache Kafka or any other endpoint.
- In case of Nuclio as a part of MLRun framework, the collected information refers to the input and data properties only and is streamed to an internal platform component that collects the data into a log dataset ready for further analysis.

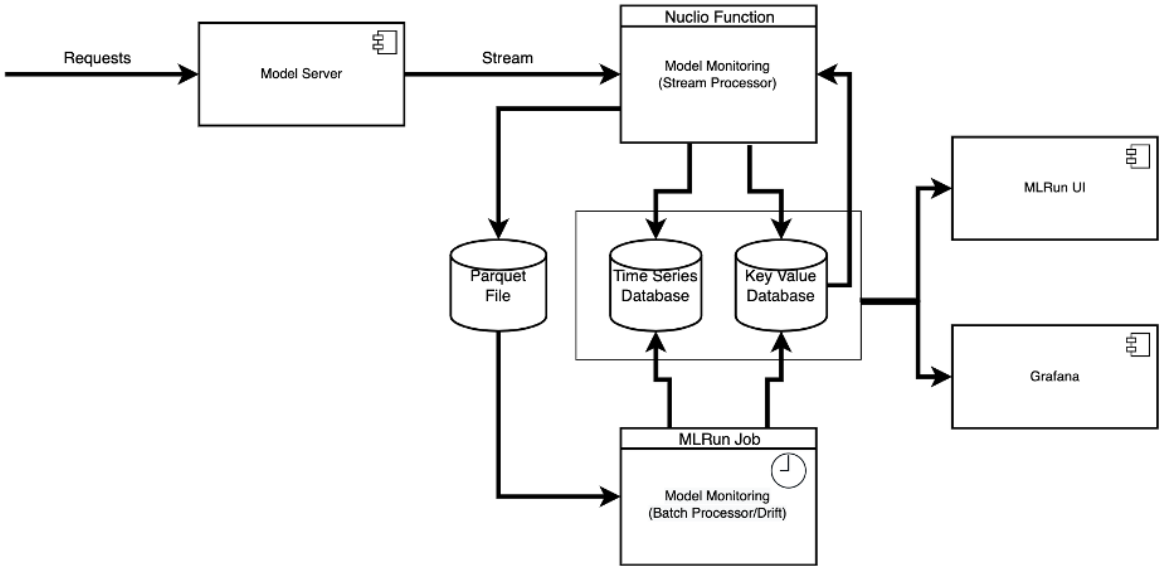


Figure 2 MLRun Model monitoring architecture

Serverless Model Monitoring

Once the data collection is enabled, the process of the model monitoring may be implemented and integrated in the platform. The monitoring architecture proposed by the MLRun framework can be easily generalized for an arbitrary implementation of the AIF life-cycle management. More specifically, the following steps are necessary:

- Continuous information collection regarding the model performance and model data.
- Storage of the data in a data repository in order to make it available for the analysis based on certain time frame defined for the scenario.
- Analysis of data in order to detect possible deviations. It is possible to enable and make generally available a set of analysis types and algorithms, such as outlier detection, adversarial detection, or drift detection. Furthermore, for the purpose of specific AI@EDGE scenarios, it is possible to introduce new algorithms more appropriate for the specific types of data in the domain.
- Report, as a new type of metric or event, the possible deviation to enable the platform to perform the relevant management actions: re-train, re-deploy, etc.

In this way, the monitoring functionality may be completely automated:

- to enable the monitoring, the AIF descriptor specifies the information about the monitoring policies. This information may refer, e.g., to the advertised AIF metrics (e.g., function latency) that, when violated, should lead to the redeployment / migration of the function; to the drift detection policy that defines the expected algorithm, the data volume, and the required threshold to consider the deviation.
- The platform component will engage the corresponding data collection pipeline using, e.g., pre-defined Serverless function and the corresponding analysis function that will report the deviations.

Depending on the framework, the implementation may be achieved in a variety of ways. MLRun, for example, makes the whole process completely automated, using a predefined drift detection analysis based on a series of metrics (Total Variation Distance, Hellinger Distance, Kullback-Leibler Distance). In case of Seldon Core this automation is not available but may be implemented on top of an open-source project Seldon Alibi Detect[6], where different types of analysis and algorithms are defined and implemented as a Python library. Extending the library with AI@EDGE solutions and integrating the monitoring pipeline into the platform functionality may lead to a pluggable and reusable reference implementation.

2.2.2 Integration of Serverless platform in the Connect-Compute Platform

In the following section, we describe how the Serverless platforms are integrated in the CCP. First, we present the integration of the general purpose Nuclio Serverless framework, as defined in Deliverable D4.1. Second, we describe how the specific frameworks supporting the AI-related functionality (e.g., ML model serving, model monitoring) such as Seldon Core, are added to the platform using the infrastructure provided by Kubernetes.

[6] https://docs.seldon.io/projects/alibi-detect/en/stable/index.html

Integration of Nuclio Serverless Platform

We have integrated the Nuclio Serverless platform in the CCP relying on the LightEdge Runtime (Figure 3). LightEdge architecture's main component is the LightEdge Core component that manages, among others, the APIs, Accounts, Environment, Time series data. In order to implement a new southbound interface with an external platform, it gives the possibility of doing so by developing a Manager. Managers are modular components that allow the Core to receive input from other platforms, e.g., by using APIs, Web Socket, proprietary protocols. In Figure 3, the relationship among these components is shown. It can be seen how many Managers (LightEdge-specific and even further custom extensions) are integrated and registered on top of a single Core instance. Each custom extension, represented in the figure as Extra Manager, defines its own primitives, API and their handlers, management implementation, etc.

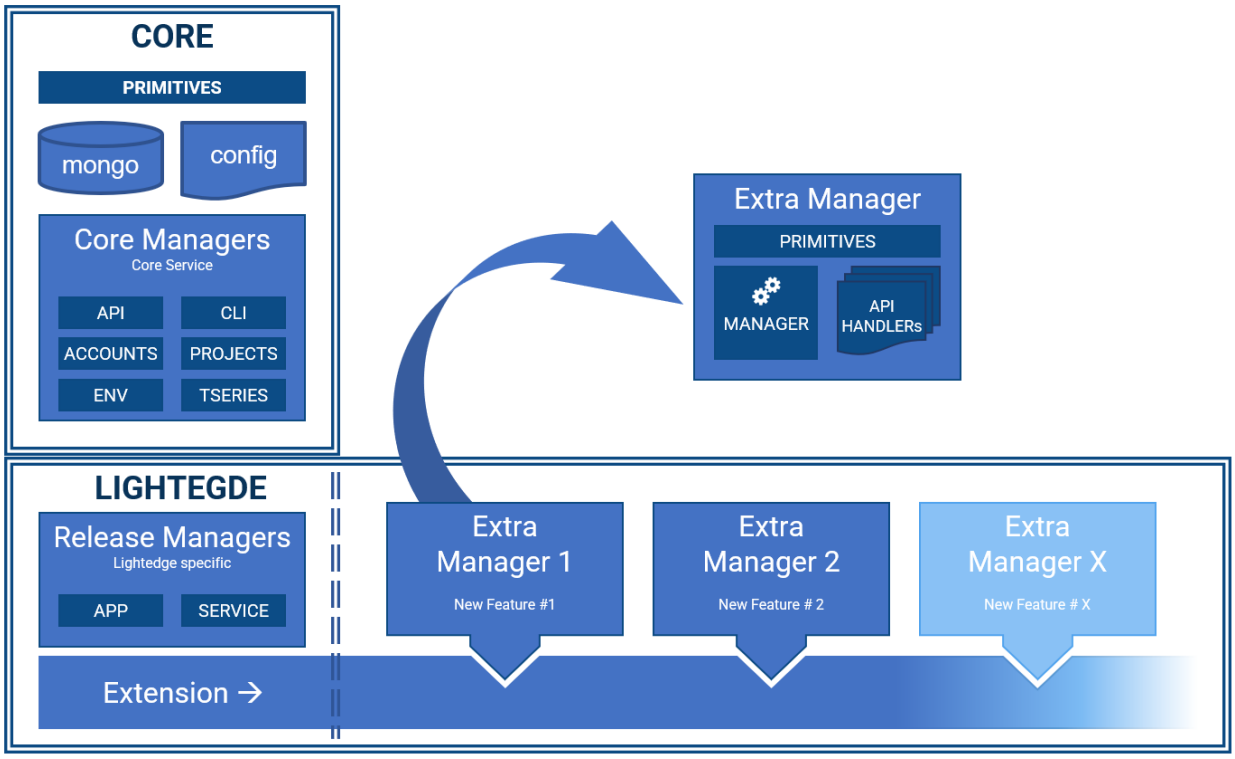


Figure 3 LightEdge runtime

Following this approach, for integrating the Serverless platform into LightEdge we have developed a specific Extra Manager, called Serverless Manager (Figure 4). This component exposes two specific APIs for Serverless platform: the function management API and the function invocation API. The implementation of the component introduces a layer of abstraction, called Serverless Platform Adapter, to support different types of Serverless platforms. Through this adapter, the component translates the requests of deploying Serverless functions arriving to LightEdge into the proper request exchange with the Serverless platform of choice. A CRUD (Create/Read/Update/Delete) API Handler manages correspondingly the requests for listing, creating, deleting, and updating the function definitions:

- GET: retrieve all/single functions
- POST: create function
- PUT: update function
- DELETE: delete function

Similarly, the API for function invocation translates using the adapter this invocation into the request for the corresponding Serverless platform function.

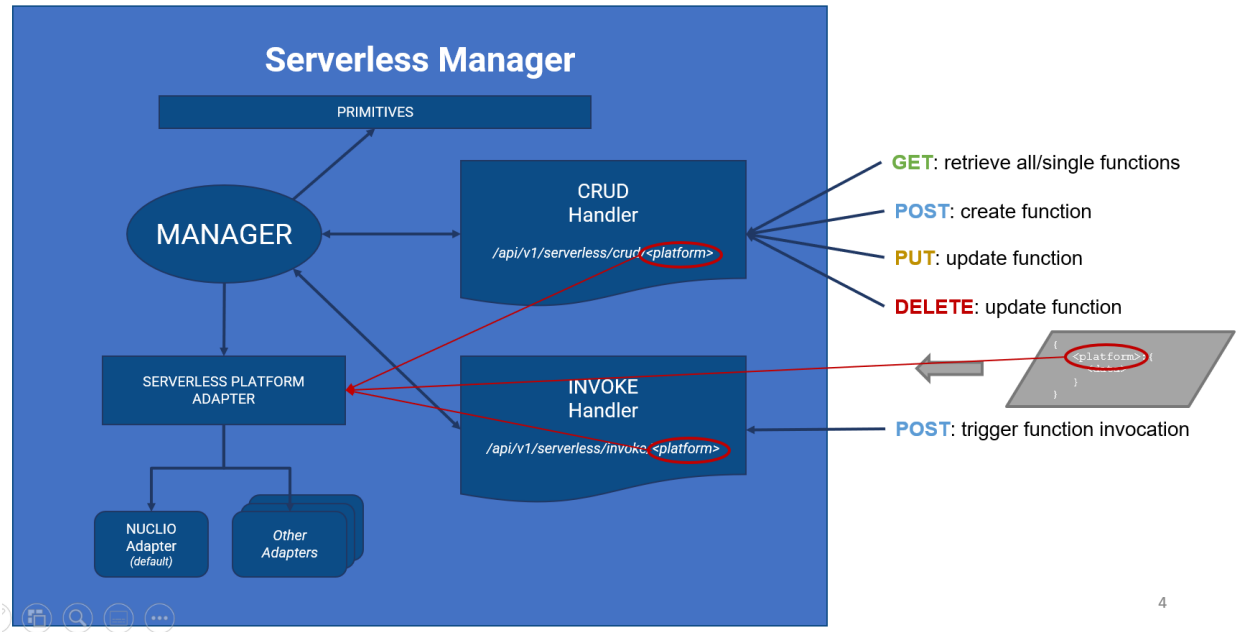


Figure 4 Serverless Manager

The Serverless Platform adapter allows handling diverse Serverless Platforms within the same Serverless Manager. It is possible to specify which platform to call at the moment of function invocation, specifying the reference to the platform in the request body. For the time being, we have developed one serverless platform adapter for the Nuclio platform. This is also the default Serverless platform if none is specified. Other adapters may be added in the future.

The content of the messages exchanged through the API of the manager depends on a specific integrated platform. Each message is represented as a generic wrapper with a content that corresponds to the format of the underlying platform. For example, the Serverless function CRUD API message format for Nuclio is

shown in Figure 5. The details of the Nuclio function metadata, function specification, and invocation structure may be found in the corresponding platform documentation[7].

url:*/api/v1/serverless/crud/<platform>*

Body:

```
{
    "nuclio":{
        <data>
    }
}
```

data (*see Dashboard HTTP API*)

GET: Specify *namespace**, *name*, *project-name*

- If *name* is specified, retrieve one functions
- Otherwise, all in the *namespace*

POST: Specify ***metadata****, ***spec****

PUT: Specify ***metadata****, ***spec, status***

DELETE: Specify ***metadata****, *validated-state*

Figure 5 Nuclio adapter

Integration of Seldon Core Framework

In many cases, the integration of frameworks like Seldon Core strictly relates to the underlying infrastructure and more specifically to Kubernetes. That is, the definition and implementation of the deployed entities (AI function model servers in this case) are achieved as Kubernetes Custom Resources. The implementation of the specific life-cycle of these custom resources is delegated to the extension of the Kubernetes Platform, called Kubernetes Operators. These operators are responsible for transforming declarative specification of the function into a deployable artifact that is then deployed, managed, and scaled by the Kubernetes platform. In case of the Seldon Core framework, the Seldon Kubernetes Operator[8] is used to manage the Serverless AI functions. Starting from a predefined model server template (or a custom one defined by the developer), the operator can create a dedicated container that hosts the AI function on top of an ML model and exposes the corresponding API.

[7] https://nuclio.io/docs/latest/

[8] https://docs.seldon.io/projects/seldon-core/en/latest/charts/seldon-core-operator.html

2.3 Disaggregated Radio Access

2.3.1 Near Real Time RIC

In this section, we describe the near real time RIC that we implemented in the scope of AI@EDGE project to support Real time RAN monitoring and closed loop control at Near Real Time granularities (10 msec to 1 sec). The reference architecture of ORAN Alliance near real time RIC is shown by Figure 6. We do work with F-release as reference Near Real Time RIC Controller, the one open sourced by ORAN alliance. This supports the development of xApps in the scope of AI@EDGE to showcase closed loop automation subsystems.

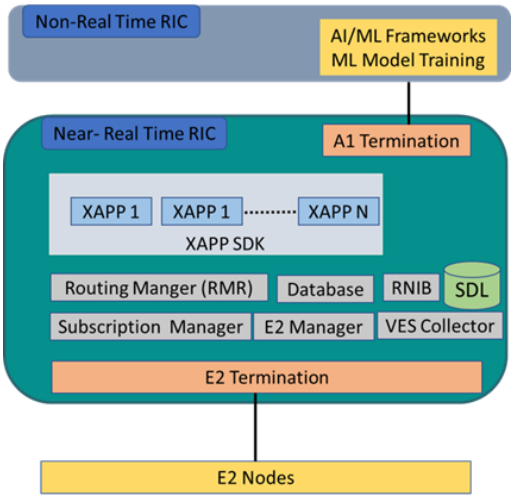


Figure 6 Near Real Time RIC ORAN Reference Architecture

Figure 7 shows the Near-Real Time RIC function cluster state from the testbed settings, with common processes; we have also benchmarked the installation with E2SIM (E2 Node Emulator provided by ORAN Alliance) and, in addition, the OSC's open source xAPP (KPI Monitoring xAPP, Bouncer xAPP) that we used to integrate pipelining and prediction tasks.

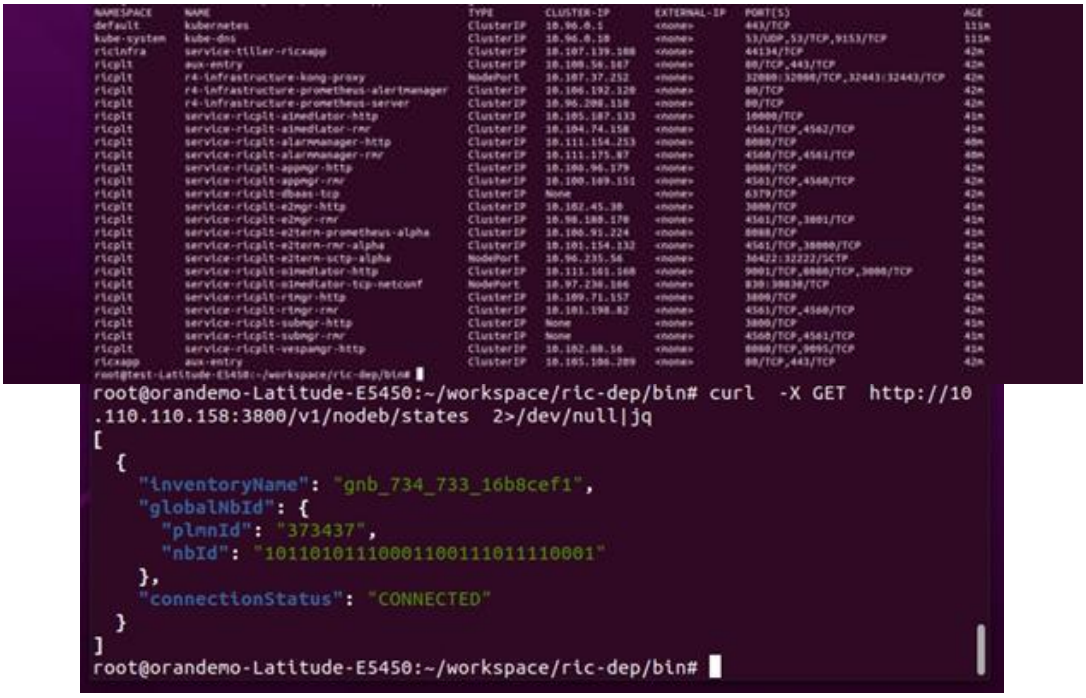


Figure 7 Near Real Time RIC Cluster and E2 Node Connection Output

2.3.2 E2 Interface

The E2 interface, developed in the frame of Task 4.3, is responsible for all communication between the near real time RIC and the RAN stack. In order to fully support this communication, we integrated the execution several protocols, specifically:

- E2 Application Protocol (E2AP) - this manages the setup and maintenance of the communication link between the RIC and the RAN stack.
- E2 Service Model (E2SM) - this manages the specific services that are provided by E2 interface, specifically the KPM (metrics reporting) the RC (the control of the RAN) and the NI (network interface message passing).

The near-RT RIC communicates with E2 nodes over the E2 interface. These nodes can be the CU-UP, CU-CP, or DU. Each E2 node exposes several RAN functions, i.e., the services and capabilities it supports. For example, DUs from different vendors may expose different control knobs to the RIC depending on which parameters and functionalities can be tuned, as well as their capability in collecting and reporting different performance metrics. By using publish-subscribe mechanics, E2 nodes can publish their data, and the xApps on the near-RT RIC can subscribe to one or more of these RAN functions through the E2 interface. This makes it possible to clearly separate the capabilities of each node and to define how the xApps interact with the RAN.

2.3.3 RAN

srsRAN

The SRS gNB Project is a full-stack software radio gNB solution for 5G networks. Implemented in efficient and portable C/C++, the software supports a wide range of baseband hardware platforms including x86, ARM and PowerPC. Running in userspace on standard linux operating systems, the SRS Project gNB can be used with any third-party core network supporting standard interfaces. The srsRAN Project code base is developed by Software Radio Systems (SRS).

The srsRAN gNB project has been modified in line with the needs of the project to support communication with the non-RT RIC over the E2 interface. This involves creating an E2 agent within the srsRAN gNB project stack and enabling srsRAN gNB Project to report specific metrics across this interface to the RIC and xApp(s).

Implementing the E2 Agent

To support the E2 agent, ASN1 packing and unpacking was added to srsRAN gNB project for each protocol. This packing/unpacking functionality was then used to create a wide range of E2 specific procedures (E2 Setup, E2 reset, RIC subscription, RIC indication etc.) These procedures are used to create the interface and use it to communicate metrics & commands between the RAN and the RIC.

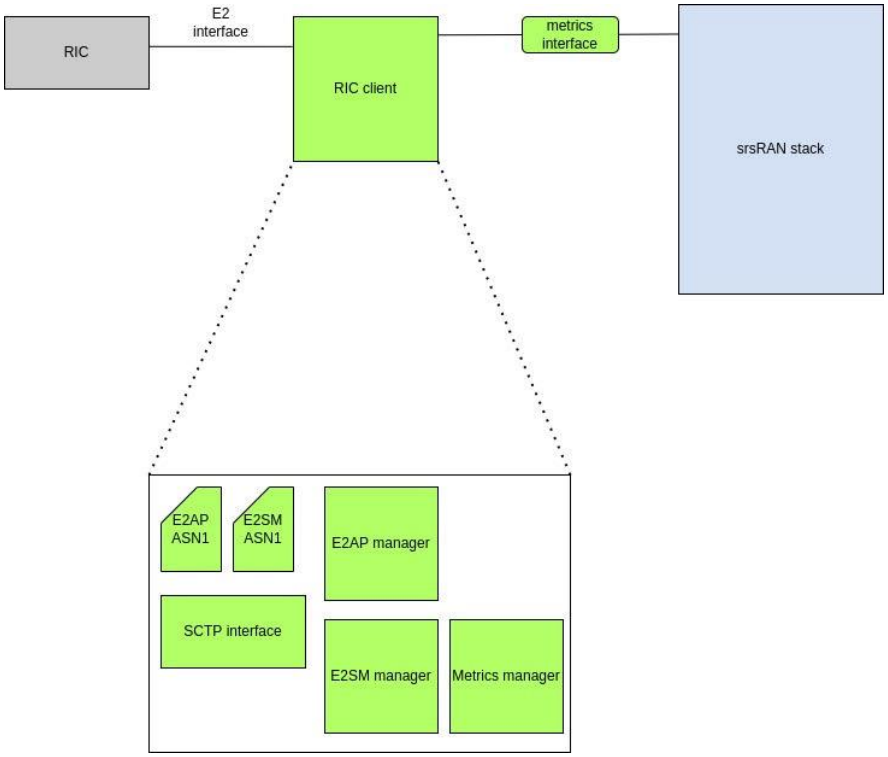


Figure 8 RAN implementation

Figure 8 outlines the codepoint blocks implementation. The interaction between the existing srsRAN stack and the near real-time RIC is managed by a RIC client which serves as the E2 agent. The RIC client interfaces with the srsRAN stack through a metrics interface, and with the external RIC via the E2 interface. The RIC client is responsible for managing the connection with the RIC, parsing the messages and extracting the required metrics from the RAN.

This RIC client is broken down into sub-components:

- E2AP ASN1 & E2SM ASN1: extensive packing and unpacking functions that allow the RIC client to serialize/deserialize hierarchical messages for sending/receiving over the E2 interface.
- E2AP manager: this module processes incoming E2 messages and creates outgoing messages, additionally it passes internal fields of messages to the E2SM manager where appropriate.
- SCTP interface: this module creates and manages the SCTP connection over which the E2 is carried.
- Metrics manager: this module converts the RAN metrics into the correct format for the E2SM manager to process.

2.3.4. Non-Real Time RIC

Figure 9 depicts the architecture of the non-RT RIC implemented in the scope of AI@EDGE project. The main focus of the developed architecture is to expose Data Exposure and RAN Control services to the rApps in order to allow intelligent automation at the RAN level in multi-RAT networks. As shown in the figure, rApps are hosted in the non-RT RIC Kubernetes cluster and have access to the Data Exposure and RAN control subsystem functions through the R1 interface, implemented as a REST API.

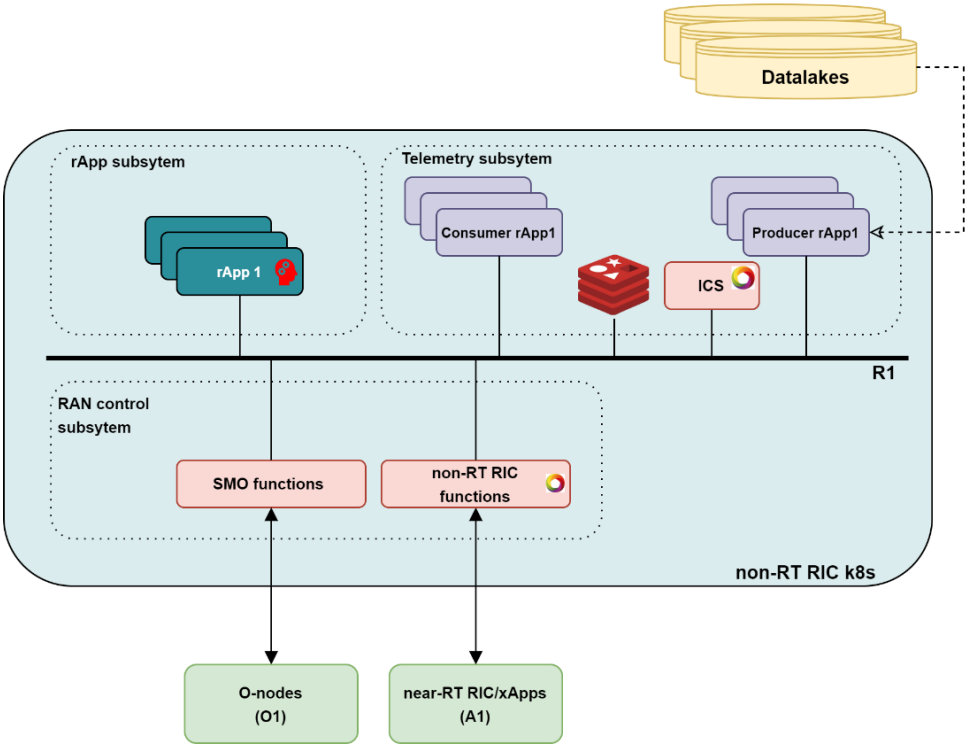


Figure 9 Non-Real Time RIC Architecture

The Data Exposure subsystem leverages the Information Coordination System (ICS) implementation from the O-RAN Software Community (OSC)[9], which provides a data subscription service decoupling data producers and data consumers. Data or Information types are defined in the ICS and can be served by one-to-many producer rApps, which are also registered in the ICS. A consumer rApp interested in one or several data types creates a data subscription through the ICS, which manages the available producers to start serving the data. The exposed data can be RAN telemetry from the O-nodes or E2-nodes, but also external data (e.g., from the NSAP) or new data produced by rApps (e.g., AI/ML-based predictions); indeed, our

9 https://docs.o-ran-sc.org/projects/o-ran-sc-nonrtric/en/f-release/overview.html#information-coordination-service

implementation includes a REDIS database to store relevant data produced by rApps and expose it in the cluster for a more efficient access. Figure 10 shows exemplary screenshots of the ICS panel during a benchmark test involving several consumers and producers.

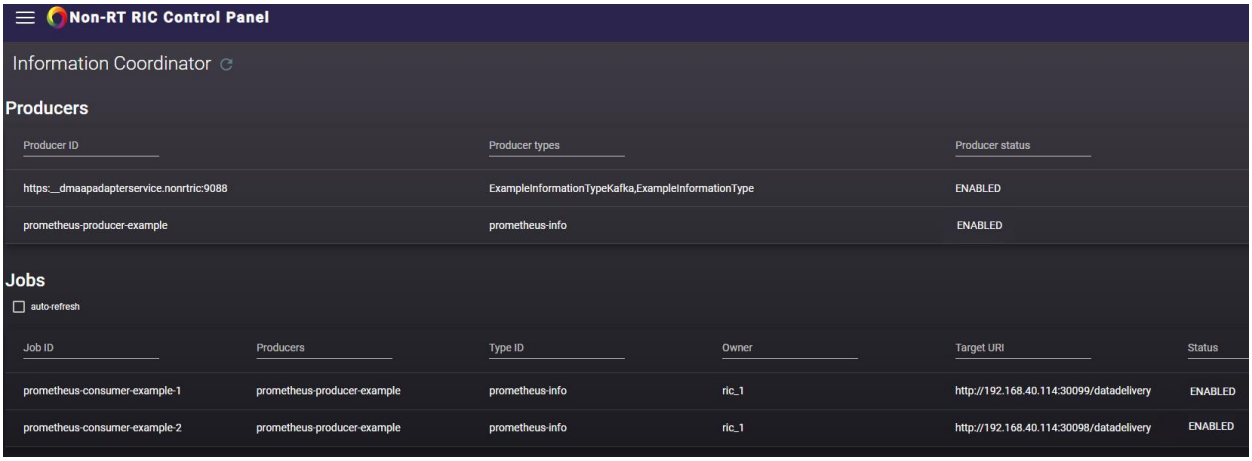


Figure 10 Non-Real Time RIC ICS Cluster and panel

The RAN Control subsystem allows rApps to manage O-nodes through the O1 interface, which has been implemented using the NETCONF protocol. Through this subsystem, rApps perform non-RT automated control of heterogeneous RAN resources in multi-RAT deployments. As will be described and evaluated in the following sections, the current implementation supports 5G, 4G and Wi-Fi technologies. Regarding the control of the near-RT RIC and the xApps, the implementation integrates the A1 Policy Management Service and Adapter from OSC[10]. Combined with the ICS, the A1 interface also allows to expose Enrichment Information to consumer xApps.

2.4 Data-driven Service Lifecycle for AI-enabled Applications

The overall life-cycle of the AIFs has been defined in the AI@EGE project from two perspectives, AIF orchestration and AIF close-loop management. More specifically, in Deliverable D2.2 the orchestration of the AIFs is defined through the steps necessary to arrive from the constructing elements (e.g., AIF code, dependencies, resources) to the complete specification of the descriptors necessary for the deployment. These steps, described in 3 phases, include in general the analysis and validation of the AIF and its dependencies (AIF graph), the definition of the configuration properties for the specific environment (possibly involving autoconfiguration), and the preparation / generation of the descriptor files necessary for

[10] https://docs.o-ran-sc.org/projects/o-ran-sc-nonrtric/en/f-release/overview.html#a1-interface-near-rt-ric-simulator

the deployment using the MTO / MEO components. Once these artifacts are ready, the onboarding of the AIF is managed by the orchestration components and the underlying VIM.

Once the AIF is deployed, its management is similar to those traditionally used for the ML solutions (MLOps). More specifically, the activities referring to the execution / operation of the AIF include monitoring of the managed solution to ensure that the solution fits the execution context and, if necessary, require the update of the solution in order to adjust to the changing context. In the first case, a typical approach for the AI/ML-based solutions is to monitor the correspondence between the model and the processed data, continuously measuring and evaluating the properties of the data processed by the solution. This approach, referred to as data drift (or model drift), allows for detecting the degrade of the quality of the applied model and therefore to trigger the need for the model update. In the second case, different techniques for the model evolution may apply based on the specific situation and the problem in hands. This evolution may include, in particular, the following scenarios:

- Complete re-training of the underlying model using the data available in the specific execution context. To support this scenario, the life-cycle may need to rely on the supporting data infrastructure, namely Data Pipeline, defined in project Deliverable 3.2 [3]. It may be also necessary to provide the corresponding training AIF that allows for producing the new model or a new version of the target AIF. To make these aspects explicit, the AIF descriptor specification allows for capturing the type of the AIF (i.e., inference and training) and the supporting data dependencies (i.e., AI/ML model reference, data to use, etc.). With this scenario, the AIF management amounts to producing a new model and to deploy a new version or to update the existing deployment with the new model.
- Incremental / continuous training of the model. In this case the model is being constructed using the continuously changing data providing a better fit to the execution context. The new versions of the models may be applied continuously, even without using the monitoring functionality.
- Replacement of the model with the predefined ones, using the characteristics of the processed data as a driver. In this case, different execution contexts may be captured with different pre-trained models and an intelligent automated algorithm may allow for their selection given the characteristics of the current settings. More specifically, analysing a slice of data corresponding to the recent AIF invocations it is possible to select the model that fits better to this data and trigger the AIF redeployment / update.
- Using the AI/ML adaptation techniques, like e.g., transfer learning, where the existing model is customized to the changing environment, without necessarily re-training the model from scratch, and therefore saving time and computation resources for such a task.

Without loss of generality, we can consider the solutions for the model evaluation as “training AIFs” as they provide (either continuously or on demand) the new versions of the “inference AIFs” to be deployed. Figure 11 represents the overall life-cycle of AI functions.

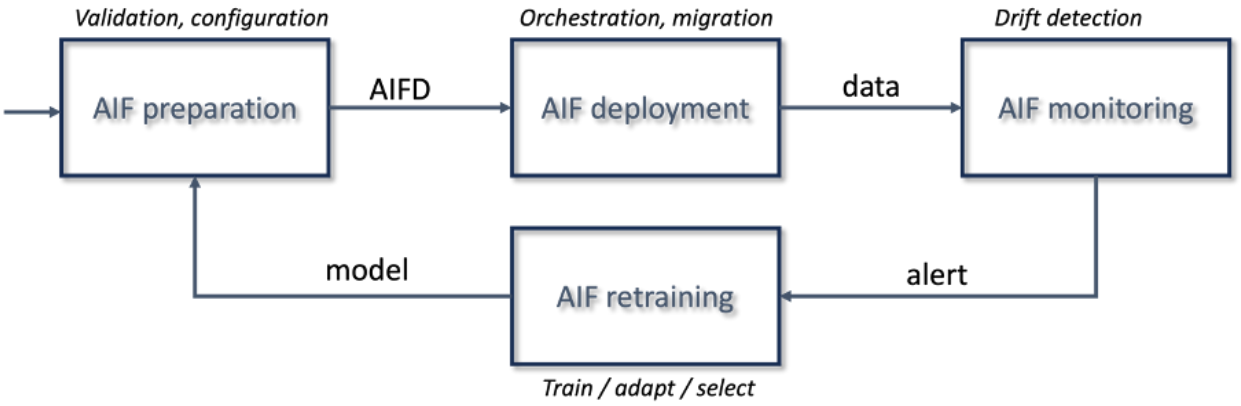


Figure 11 AIF generic life-cycle

The implementation of the different phases in general may be done in various manners. Deliverable D3.2 defines the implementation of the centralized component, Model Manager, that in collaboration with Data Pipeline is responsible for the monitoring, evolution, and preparation of the new AIF deployment. As a complement, some of these functionalities may be realized in a more distributed and autonomous manner:

- As it has been described in Section 2.2, a typical scenario for the deployment of ML-based inference functions may be achieved with the AI-specific Serverless solutions, such as Seldon Core, where the AIF server container remains the same and the only change refers to the model artifacts, which is made available through e.g., a model registry. In this case the AIF preparation is reduced to the configuration of the new model version reference.
- AIF monitoring may be "embedded" into the AIF execution container and include all the necessary elements to perform the monitoring and evaluate the model quality.
- AIF retraining/evolution may be performed on demand, deploying the corresponding function to train/create a new version and scaling to 0 when this is done. The trained function, in compliance with the corresponding Serverless framework is published to the model registry in a standard form.
- AIF preparation and upgrade may also be "embedded" into the function itself. That is, the function may monitor the registry to detect the new version of the model and to update itself to work with this new version, without any execution downtime necessary for the artifact redeployment.

In the following, we present in detail the solutions based on the Serverless framework for the model monitoring and upgrade, as well as the AIF auto-reconfiguration for the selection of suitable models without complete retraining.

2.4.1. Monitoring: Model monitoring solution with Sidecar and Seldon Alibi Detect

In order to augment the lifecycle management of models deployed via AIFs at the CCP level, this subsection introduces a drift detection functionality. Monitoring models in a production environment is crucial as these models can be sensitive to variations in data distribution, leading to a decline in model performance and

generating inaccurate outcomes that impact the quality of service (QoS). This study focuses on monitoring the model to detect cases of drift.

Drift phenomena occur when the underlying distributions of test data X and Y differ from the training data X' and Y'. In such cases, it can no longer be ensured that the model's performance with production data will be equivalent to the performance achieved during the training stage. Although various types of drift exist[11], this study concentrates on covariate drift, specifically analysing input inference data.

To facilitate model monitoring in the context of AI@EDGE, the key idea involves integrating this functionality into the deployment of AIFs in such a way the detectors are treated as additional models which receive input inference data. To showcase this, a particular AIF called Model Monitoring AIF (MMAIF) is shown in Figure 12, as well as its sub-components and interactions with other blocks. These blocks are software components containerized through Docker [17], the components of MMAIF are wrapped up altogether using Helm charts [18] and deployed within the MEC system based on a k8s cluster. The exception in this case is the Training AIFs (TAIFs), which are essentially AIF descriptors triggered by the MTO. These are included in Figure 12 to showcase the complete closed-loop system.

[11] https://docs.seldon.io/projects/alibi-detect/en/latest/cd/background.html

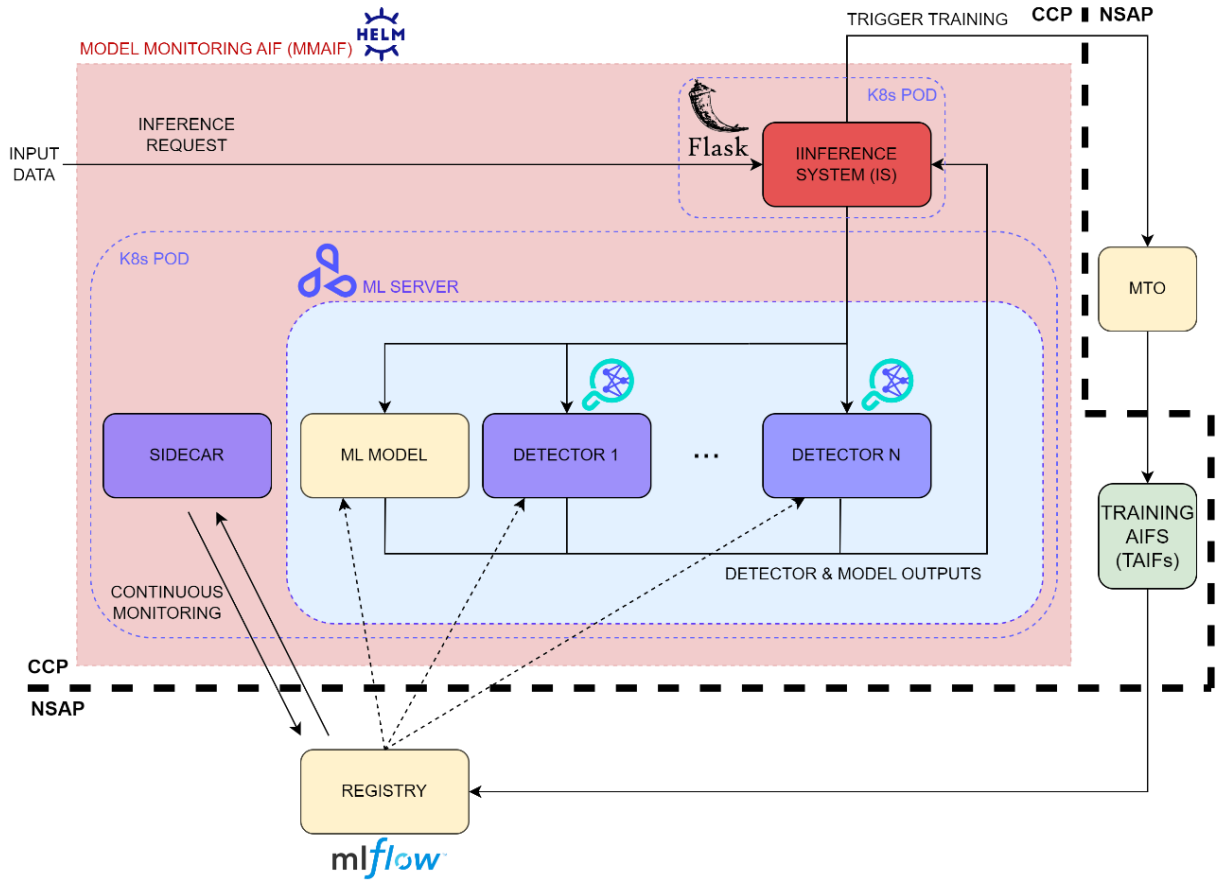


Figure 12 Monitoring & retraining AIFs closed loop

This section will first describe the new blocks introduced in the MMAIF, followed by an explanation of the functional behaviour of the entire workflow to provide insights into the closed-loop behaviour.

To align with the AI@EDGE architecture, a clear distinction is made between the CCP and the NSAP, represented by a large black dashed line. The NSAP includes the MTO and the Model Registry, as depicted in Section 2.8 and in Deliverable D3.3, which are used for AIF deployment and model storage/versioning purposes, respectively.

Within the CCP, the MMAIF represents a specific AIF that encompasses a model and its corresponding drift monitoring. As shown in Figure 12, the ML model and detectors are served using the MLServer [19], an open-source inference server developed by Seldon Core [20] that supports the standard V2 inference protocol on gRPC and REST Flavors. Since AIF deployment is performed through k8s, a sidecar is deployed alongside the inference server to continuously monitor the availability of newer versions at the MLFlow registry NSAP component, as explained later in section 2.4.3. In this case, the sidecar not only monitors the model but also the detectors. These detectors are constructed using Alibi Detect [21], an open-source Python library for drift, outlier, and adversarial detection. As MLFlow and Alibi Detect lack compatibility for storing detectors as models in MLFlow registry, these are uploaded as artifacts and stored

in MinIO [22]. Lastly, the sidecar also monitors the MinIO instance for new versions. The final block within the MMAIF is the Inference System (IS), developed using Flask [23], which acts as a middleware in order to send the inference requests to both, the model and the deployed detectors, and forwards the model's inference output outside the AIF. Additionally, the IS serves as an "alerting system" by parsing the output of the detectors to determine the appropriate subsequent actions. Instead of relying on Seldon Core to perform inference from outside the k8s cluster, the input data is directly sent to the IS that exposes separate endpoints, one for each model/detector in the MLServer component.

Regarding the functional behaviour, the whole closed loop behaviour is explained as follows, from the injection of input inference data to the execution of the Training-AIF function. Firstly, the input inference data is sent directly to the IS via REST endpoints for the model and detectors. Once the IS receives the data, it forwards it to the model and detectors using REST endpoints exposed by the MLServer. After the inference process concludes, the IS sends the model output outside the cluster to deliver the model response, as for the detectors' output, the IS parses the response, leading to two potential scenarios. If no drift is detected, no further actions are taken, and the IS sends the detector's reply outside the cluster along with the model output. In the case of drift detection, the IS calls the MTO via a REST API to trigger the execution of the Training-AIFs block. This initiates the retraining of the model and detectors, which are subsequently uploaded to the registry once the execution is complete.

2.4.2 Re-training: Model Autoconfiguration

To enhance the lifecycle management of models deployed through AIFs at the CCP level, this subsection introduces the auto-configuration feature.

In this subsection, we specifically refer to the configuration process as “hyper-parameter optimization” (HPO) for an AIF. The primary objective is to identify the best hyperparameter configuration for the AIF, thereby enhancing its performance on a given dataset.

For instance, when using a Random Forest classifier as an AIF, certain Hyper-Parameters (HPs) such as the minimum number of samples required at a leaf node and the number of features to consider for the best split significantly affect the classification performance.

Hyper-parameter optimization (HPO) involves fine-tuning these hyper-parameters to achieve the highest level of performance. Widely used HPO techniques, such as Grid Search and Bayesian Optimization, systematically explore the hyper-parameter space to identify the optimal configuration. By selecting the most suitable hyperparameter values, these approaches aim to enhance the AIF's capabilities and yield improved results in various tasks.

While HPO is essential for achieving optimal performance, it comes with its challenges. One of the significant issues is the computational and time costs associated with these optimization techniques. As the search for the best hyper-parameter configuration starts from scratch for each new dataset, it can be resource-intensive and time-consuming, limiting the practicality of frequent reconfiguration. In response to the challenges posed by hyper-parameter optimization and adapting ML, we propose a novel approach to enhance efficiency and adaptability. This approach harnesses the power of meta-learning, which involves leveraging prior configuration optimization experiences to generalize the configuration across diverse

datasets. By drawing on knowledge from past optimization tasks, meta-learning aims to expedite the search for efficient hyper-parameter configurations and facilitate more frequent reconfiguration without compromising performance.

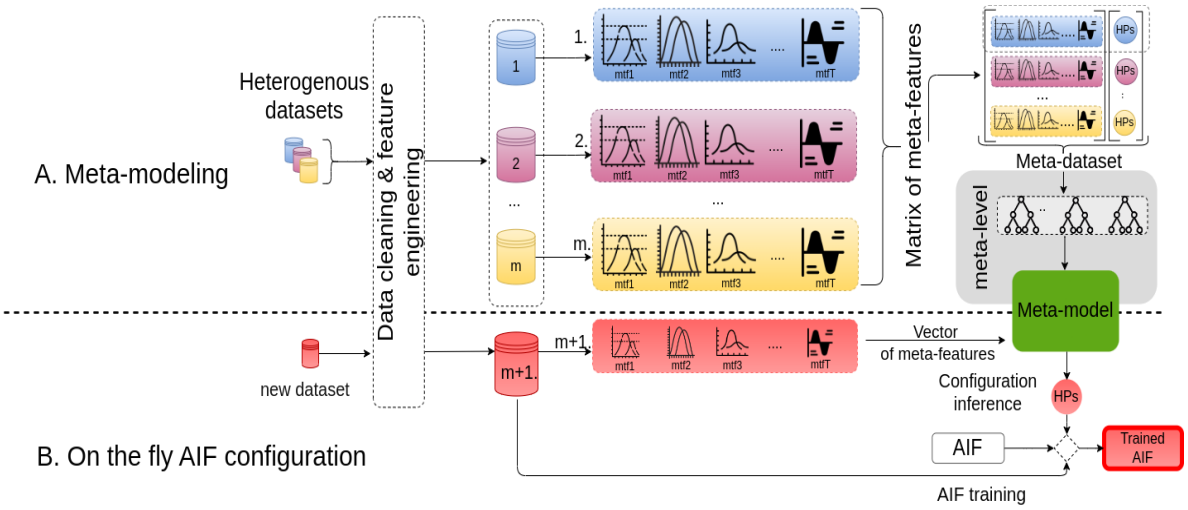


Figure 13 Initial meta-modeling and on-the-fly configuration phases

In our proposed approach, we aim to optimize the AIF configuration across diverse contexts using Hyper-Parameter Optimization (HPO). We then leverage meta-learning to make efficient use of these past experiences.

As depicted in Figure 13, our approach involves an initial meta-modeling phase where we construct a meta-dataset comprising two essential components. Firstly, we use 36 carefully selected meta-features, which provide a comprehensive set of characteristics from diverse and heterogeneous datasets. These meta-features are designed for fast computation and are independent of the nature of the underlying data.

Secondly, the meta-dataset includes information about the corresponding optimal AIFs configurations used for each of these datasets. To obtain this information, we employ a hyper-parameter optimization (HPO) solution to find the best configuration for each context network.

Using the constructed meta-dataset, we then train a regression model called the meta-model. This meta-model learns from past experiences and can predict the optimal NIDS configuration for new datasets.

In the on-the-fly AIF configuration phase, when a new dataset is collected, the AIF can be quickly configured by leveraging the trained meta-model to infer the best hyper-parameter settings. This approach yields detection performance comparable to traditional HPO techniques, but with a significant reduction in resource usage. On average, our method is nine times faster, demonstrating its efficiency and effectiveness in optimizing the AIF configuration for different network contexts.

2.4.3 Autonomous model update with Sidecar and Model Registry

We propose a sidecar architecture of model services to facilitate automated and autonomous model updates in the AI@EDGE context. The main goal of automated model updates is to ensure that the model is always up-to-date and accurate, improving AI@EDGE performance and providing the most accurate predictions possible. Automatic updates can also save time and resources by eliminating the need for manual updates and reducing the risk of human error.

The architecture of the automatic update model is illustrated in Figure 14.

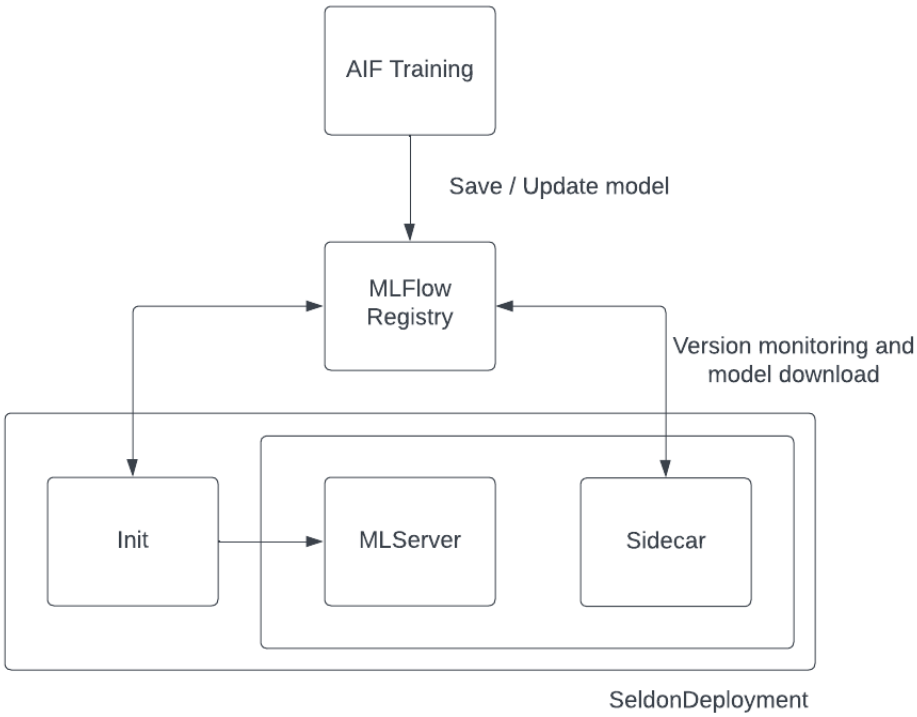


Figure 14 Sidecar architecture for autonomous model update

The technology stack used to implement the architecture is as follows:

- Kubernetes as the VIM implementation adopted by AI@EDGE
- Seldon Core as the ML Model Serving Serverless infrastructure.
- MLFlow (plus MiIO and Postgres) as the model registry.

As described in Section 2.2, Seldon Core is an open-source platform for deploying and managing machine learning models on Kubernetes. It provides several tools for MLOps, including model serving, monitoring, and scaling. MLFlow is an open-source platform for end-to-end lifecycle management of machine learning, including model training, deployment, and monitoring. In our case, we used the model registry it provides. We also included MinIO and Postgres in the stack, which allow users to store and manage machine learning

models in a centralized location. Together, these components provide a powerful platform for managing and deploying machine learning models at scale.

With this architecture, we can achieve closed-loop automation within the CCP regarding the automated redeployment of models. At a high level, the auto-update lifecycle works as follows:

- A new version of a model is trained and registered in MLFlow, following the MLFlow model format for persistence. This allows the model to be reused in a variety of downstream tools, for example with Seldon Core and MLServer (the tool used by Seldon to deploy a model as a service).
- Once the model is saved, we pass its URI reference to a Seldon Deployment CRD packaged in a Helm Chart. Using Helm allows us to distribute the AIF through the CCP (MEO + LightEdge).
- Within the Seldon Deployment, we inject a sidecar application. This sidecar constantly monitors the status of the model versions within the MLFlow registry. When it detects a new version of the model, it downloads it to a folder shared with the Seldon container and reloads the service with the new model. In this way, there is no downtime due to model redeployment. Furthermore, there is no need for the generation of a new image for the AIF container and therefore the overall time from the model creation to serving is drastically reduced.

The information about the model is provided as a part of the AIF description.

2.5 AIF Descriptor

The AIF Descriptor (AIFD) is a deployment template used by the CCP for the management of the AIF life-cycle. It aims at supporting the different orchestration components in making decisions about resource allocation, placement, dependencies, monitoring, etc. In the following subsections, different components of the descriptor are presented. With respect to the initial model presented in Deliverable D4.1, this document provides a detailed model of the AIF-specific extensions. The complete specification is provided in Annex1.

2.5.1 Relevant existing Descriptors and their relevance for AI@EDGE

ETSI MEC *Application descriptor information model (AppD)*

The MEC application descriptor (AppD), including its attributes, is defined in ETSI GS MEC 010-2 [31]. It may be included in a MEC application package, encoded e.g., in TOSCA or YANG format.

MEC application specifies in its descriptor (AppD) [31] some MEC-specific fields, such as the maximum tolerated latency, the set of required MEC platform services, traffic rules that allow to redirect the traffic to the MEC application, and the preferred deployment location. An AppD is a part of the application package and describes application requirements and rules required by the application provider. Specifically, traffic steering rules comprise of traffic filters to identify the packets, actions to be taken, and the destination interface to receive the packets (the MEO passes the traffic steering rules in the AppD to the MEC Platform of the chosen MEC host). The ServiceDescriptor data type describes in AppD a MEC service produced by a service-providing MEC application.

HELM Charts

Helm is a commonly used Kubernetes package and operations manager, it uses helm charts as packaging format, based on yaml. A chart is a collection of files that describe a related set of Kubernetes resources: they contain the declarative Kubernetes resource files required to deploy an application. It can also declare one or more dependencies the application needs to run. A single chart might be used to deploy something simple, like a memcached pod, or something complex, like a full web app stack with HTTP servers, databases, caches, and so on. Helm charts are used to deploy an application, or one component of a larger application. Charts are created as files laid out in a particular directory tree. A Helm chart can contain any number of Kubernetes objects, all of which are deployed as part of the chart. A Helm chart will usually contain at least a Deployment and a Service, but it can also contain an Ingress, Persistent Volume Claims, or any other Kubernetes object. They can be packaged into versioned archives to be deployed.

OSM Information Model (VNFD)

ETSI Open Source MANO (OSM) Information Model defines how the Virtual Network Functions (VNF) are configured and deployed. OSM fulfils the role of NFVO in ETSI NFV architecture. While the modeling of CNF or any Kubernetes applications has not yet been included in ETSI NFV SOL006, OSM Information Model allows for defining Container Network Functions (CNF) using one or more KDUs (Kubernetes Deployment Unit). Specifically, KDUs declare the helm chart/s, connection points (mgmt-ext) where Kubernetes services of this helm-chart are exposed, and certain k8s-cluster requirements.

ONAP Application Service Descriptor (ASD) and packaging Proposals for CNF

Open Network Automation Platform (ONAP) uses Application Service Descriptor (ASD) that defines an information model for packaging the network functions and applications providing the necessary information regarding installation, provisioning, and tooling. Packaging proposal for CNF specifically defines the minimum information for the orchestrator, and pointers to cloud-native artifacts and code (including configuration) required for the LCM implementation in the Cloud environments and platforms, in particular on Kubernetes. Helm Charts are the primary deployment artifact for a containerized application.

2.5.2 AIF Descriptor Information Model definition

In the following we define the definition of the data structures to be used by AIF descriptor information model. An AIF Descriptor (AIFD) is a specification of the AIF, which describes the rules and requirements for the provisioning and management of the AIF module. It is provided as a part of the deployment package (together with other files).

The model is structured into domains and modules (Figure 15) to differentiate between different types of information elements and their use. A core model provides generic information elements for the AIF specification. In order to provide an end-to-end model view, it is possible to federate Information Models from different sources (MEC, NFV, etc.).

When deployed in the MEC host, the AIF is deployed as a MEC Application, and the AIF descriptor will describe the rules and requirements of the AIF module as a MEC Application [31]. For this reason, the AIF Information Model will consider as a basis the MEC App Information model and therefore the AIF package contains also all the information required by a regular MEC application package (as described in [31]). The MEC application package unifies the MEC package format for both the classical and MEC NFV deployment cases. As such, it is defined in such a way that the package can be both used by the MEAO directly and can be on-boarded as a VNF package to the NFVO without any change.

The **AIF package** is a bundle of files provided by AIF provider, to be on-boarded into MEC system and used by the MEC system for AIF instantiation. It typically includes the AIFD, a software image (in this case a container) or a URI to a software image, and a manifest file. It can contain also other optional files. The AIF package contains the necessary information of an AIF, used by the MEC system for AIF lifecycle management. To facilitate the integration with the orchestration tools, the AIF application package format is aligned with the MEC application package.

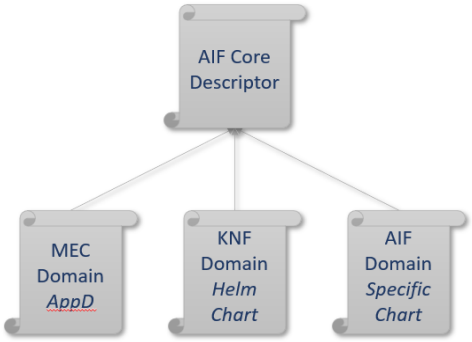


Figure 15 AIF Descriptor domains

As specified above, AIF Information Model will consider as a basis the MEC App Information model. The MEC Application descriptor, the AppD, potentially includes both MEC App LCM management information (e.g., it describes CPU, memory, hardware acceleration requirements; the reference to the virtual image of the MEC App; etc.) and MEC application configuration parameters, to be enforced by the MEC Platform. These include the description of MEC services:

- services a MEC application requires to run.
- services a MEC application may use if available.
- services a MEC application is able to produce to the platform or other MEC applications. Only relevant for service producing app
- features a MEC application requires to run.
- features a MEC application may use if available.
- Transports, if any, that this application requires to be provided by the platform. These transports will be used by the application to deliver services provided by this application. Only relevant for

service-producing apps. This attribute indicates groups of transport bindings which a service-producing MEC application requires to be supported by the platform in order to be able to produce its services. At least one of the indicated groups needs to be supported to fulfil the requirements.

The AppD also reports the information necessary for handling the traffic and implementing the route of IP packets to MEC applications:

- The traffic rules the MEC application requires.
- The DNS rules the MEC application requires.
- the maximum latency tolerated by the MEC application.

Since the AIF is deployed in the MEC system as a MEC App, the AIF descriptor should also allow to report the same information.

2.5.3 AIFs' features and attributes (relevant to the AIF descriptor)

The MEC application descriptor specifies the attributes that need to be provided for the MEO (mobile edge orchestrator) to deploy the MEC application, and therefore the specification also applies to the AIFs. However, an AIF application is qualitatively different from a generic MEC application as it has an AI function which brings with it an additional set of characteristics to define for deployment and management. These attributes should consider different types of AI functions implemented and should be applicable to a wide range of AIFs. In the following subsections we list additional AIF specific attributes that need to be provided to the MEO to enable the orchestration of AIF MEC applications. An important requirement for these AIFs is the ability to deploy them in a distributed manner for federated learning. The federated learning paradigm requires a communication/transport channel between distributed AIFs to exchange data and model parameters throughout the lifecycle of learning. These channels need to be defined with the required constraints for each specific AIF application. Several AIFs also have requirements on the properties of the data sources that serve as their input. Time granularity and probabilities associated with predictions are some examples. The output from an AIF is also subject to some additional requirements that are necessary when their output is used as an input by other AIFs. Declaration of dependencies is important to identify loops where an AIF (A) could be using, as input, predictions from another AIF (B) that used AIF (A)'s output as input. AIFs also need runtime attributes, such as a debug feature for debugging and a data augmentation feature for model updates (or retraining). In addition, the AIF should have an attribute that provides information of the actual model used. Considering all the points discussed above we have consolidated the additional attributes with a short description below.

2.5.4 AIF Deployment Information

As a part of the AIF descriptor, it is necessary to provide LCM information for the function's implementation and deployment. It is important to note that the same function may require different alternative deployments corresponding to different available infrastructures. This is the case, for example, when the AI function may have different implementation for the different target acceleration platforms and the infrastructures available. Such deployments, therefore, come with their own artifacts (e.g., different

images and/or charts), different capabilities (e.g., KPIs), and different requirements (e.g., different clusters characteristics, computation environments, data requirements).

To support such many options, it is proposed to define the LCM information through different deployment profiles, with their own characteristics and descriptions. Each profile exposes the information about:

- Required environment: platform capabilities (cluster properties, plugins, node properties), computation properties (CPU/GPU, RAM), etc.
- Advertised metrics and KPIs: the (range of) values that can be achieved with the profile.
- AIF requirements: specific data properties and requirements

Furthermore, considering Kubernetes as the target VIM, we expect the deployment profile to define the Kubernetes Deployment Unit (KDU) information as a main solution for the LCM information provisioning.

Given the fact that the function potentially defines more than one deployment profile, it is the responsibility of the orchestrator to make the selection based on:

- The verification of the available infrastructure against the profile requirements
- Optimization of the resources and KPIs based on the profile requirements and advertised metrics. Such optimization may potentially use some intelligent AI-based mechanisms.

2.5.5 AI@EDGE AIF descriptor based on (or augmenting) AppD

The Descriptor specification is detailed in Annex 6.1. It is structured in the following subsections, corresponding to different aspects of the description as defined above.

Base AIF metadata. The base AIF descriptor relies on the attributes of the MEC AppD and defines the metadata information for the function, such as the function description and identity, used and exposed services, features, etc.

Deployment Profiles. The deployment profiles section defines the potential LCM configurations for the same function. Each deployment profile is defined with the Kubernetes Deployment Unit structure characterizing the deployment requirements and the deployment specification necessary for it. Note an important feature for the AI function deployment characterizing the required acceleration hardware and the AIF metrics that can be achieved with the acceleration profile applied. This information is fundamental for the placement and migration decisions performed by the relevant AI@EDGE platform components.

2.5.6 AI Specific Features

The functionalities specific to AI functions are defined with the specific AIF descriptor sections that define:

- the requirements related to the deployment of AIF, that can be managed by the orchestration process.
- general information regarding AIFs with the purpose of informing users.

General AI properties. In the specification of AIF, it is important to explicitly distinguish the nature of the operation that the function performs: whether it is used as predictor or as for training. The first case defines the prediction requirements, and the second one describes the training requirements.

By "training", we mean the training phase of an ML / DL / RL algorithm. With the term "prediction", we mean the phase in which the trained algorithm generates predictions or, in general, performs inference tasks.

The separation is based on the following considerations:

- Training and prediction/inference require different resources. Typically, training requires a greater amount of hardware resources. Separating training and prediction description allows greater discretion to the orchestrator for deployment based on resource availability.
- The input and output of the two phases are different. In the training phase, usually, the input is a well-defined dataset. The output could be a set of artifacts that include the trained model, any requirements and/or a set of metadata that describe the model, the execution environment, etc. In the prediction phase, usually, the input is a set of data passed to a predictor function. The output is a generated prediction, numerical or categorical. Sometimes, the output includes a set of metadata that describes the prediction.
- The training phase requires a series of additional parameterizations compared to the prediction phase. For example, it requires specifications for the training mode (centralized, distributed, or federated) or for the updating of pre-existing models.
- Making the declaration of the two aspects optional in the descriptor covers more scenarios. For example, there are scenarios that require the simple deployment of a prediction function, and others where both training and prediction are required.

Please note that the distinction may be achieved in different forms. One approach is to use different sections in the AIF descriptor that cover these two operations. In this case within the same descriptor, it is possible to completely cover all the aspects. On the other hand, it defines a generalized view on the resource usage and constraints, making it difficult to execute some potential optimizations. Another approach is to have separate descriptors for the two operations making one of the optionally depending on the other. This allows for higher resource and deployment profile declaration flexibility, also making the life cycle of the two parts more independent.

The AIF schema descriptor therefore defines which type of operation is being engaged, provides the information about the ML model involved, similar it is done in the Model Cards in various platforms (e.g., HuggingFace[12] for general AI operations), how the operation should be activated, what are the information it relies on (Input Sources) and which information it provides (outputs), the dependencies as a part of the

[12] https://huggingface.co/

inference application, and eventually the distribution information (e.g., for Federated Learning training operations).

Activation descriptor. By "activation" we mean the way AIF's operations are activated. We can consider the following activations:

- Event-based function is triggered externally, by a message or by an API call, or simply one-time when deployed.
- Time-based is triggered by an internal regular time events.
- Permanent function is activated with the deployment and remain active following its internal logic.

Event-based triggers activate the function upon the occurrence of an event. There can be three types of trigger events: API-call request, message publication on a channel AIF is subscribed to, activation on deploy.

The time-based triggers activate the function temporally, with one-time or recurring activation, through scheduling or a cron.

There are several deployment scenarios covered by these descriptors, related both to training and inference AIF phases. For example, we can have cases of permanent deployment of AIF where continuous training strategy is adopted, training/prediction activation scheduled by time or triggered by events. Typical cases for inference execution are API calls from requesters, or scheduled prediction that writes their outputs on a message channel.

Input sources. The following section describes the data input of an AIF. The data sources come from various types of services (APIs, messaging channels, data pipelines, other AIFs) and can be stored in some persistent storages (databases, object storages). The input sources characterize the information required by the AIF to function properly and to be able to perform its operations (whether predictions or training). This may include the supporting data necessary for the function to perform and adjust inference, the dataset necessary to perform training.

When we define a data source, we specify a set of parameters that an orchestrator can use and a set of metadata that enables a human understanding of the resource. The data relevant to the orchestrator are:

- A unique identifier of the data source.
- A description of the data source type.
- A list of dependencies. A resource could require the existence of other data sources or services.
- A set of requirements, related to network performance. These requirements may affect the orchestrator's ability to request the data source and deploy AIF.

We distinguish the following types of data sources:

- **Raw data**: the information regarding a particular sensed data known, exposed, and collected at the platform level. This may be, e.g., a raw metric of the underlying VIM, RAN metrics, etc. In this

case the data source type is uniquely identified by the underlying resource and its metric type. These data sources are intrinsically stream-based.

- **Storage**: the data has already been collected and stored at the platform (either at NSAP or at the MEC level). In this case the data source is described with reference to the storage and the dataset identity. A corresponding supporting mechanism available to the orchestration is responsible for ensuring proper access to the resource for the function considering, e.g., the slicing aspects, privacy, etc. The storage-type sources may represent the raw metrics collected by the underlying infrastructure (e.g., Prometheus sources) or derived by pipelines and other AIFs.
- **Pipeline**: in this way, it is declared that the AIF depends on the presence of a data source, which is implemented as (a part of) a pipeline. In other words, it must have a pipeline deployed that generates and publishes the data in a certain manner. Differently to the Raw data sources, the pipeline source describes the derived data delivered in a stream-based way (e.g., through the corresponding message broker).

The requirements we propose for the data source affect the decision of the orchestrator on how and whether to deploy an AIF:

- **dataBandwith**, is used to avoid traffic congestion or issues with data size transmission. Defines the **required** bandwidth allocated by the infrastructure to the function that is necessary to process the input data.
- **dataLatency**, declares the maximum latency **required** by AIF to avoid data stale.
- **dataConfidence**, if the data source is produced as an output of some AIF function, the confidence **should be** bigger than a specified threshold for the corresponding metric.
- **dataWindow**: the minimal time interval (absolute or sliding) or data point amount of the data collection (for storage-based sources) that **should be** made available for the function before it can perform its operations. This may represent, e.g., the minimal dataset for the training function to perform its operations.

Finally, it is possible to automate the data pipelining necessary for the AIF to work through a set of additional requirements:

- **dataAugmentation**. Defines whether and how the data augmentation **should be** engaged to provide synthetic data in addition to the original data for e.g., improving the training.
- **enableDataCollection**. If set to true, the data collection **should be** engaged by the platform together with the function deployment. Otherwise, it is expected that the data is already available in the context for the function to be deployed.
- **dataAggregation**: if specified, defines how the data originating from the data source should be aggregated before given as input to the function. Defines time intervals to group the data (e.g., 1h) and the aggregation function (e.g., avg, sum, count). The aggregator should be deployed or made already available at the moment of function deployment. This way, we allow the possibility to have,

for example, some predefined functions that perform basic operations over raw network data, like simple metric aggregation over time.

Output. An output descriptor defines the output of an AIF. The possible results of an AIF can be a file or a set of files (e.g., a trained model, model metrics, derived dataset, etc.), an inference, a set of performance of the AIF (for example time consumed to perform the task) or the confidence of the prediction (whether it is precision, recall or other ML/DL metrics). The output format can be structured with a schema (JSON or table schema) and can be redirected to a storage or a message channel.

Distribution strategy. By distribution, we mean the distribution of the workload of an AIF over one or more workers within a network. The possible cases are of three types:

- Centralized, where AIF workload is performed on a single node.
- Distributed, where the AIF workload is distributed across multiple nodes by sharing data among the nodes.
- Federated, where the AIF workload is distributed across multiple nodes but not sharing data between the nodes.

In the first case, the orchestrator deploys AIF on a single node that acts both as a server and as a client. AIF is trained on a single node that has access to the required data.

In the second case, the orchestrator deploys AIF on multiple nodes, where a central node acts as a coordinator and the other nodes act as workers. The coordinator can spread the training workload across multiple workers by sharing data. In this sense, network workers potentially have access to all data. The information exchanged between the nodes can be data or model parameters.

In the third case, the orchestrator deploys AIF on multiple nodes, where a central node can act as the training coordinator, or there may be a peer-to-peer situation. Each node has access to its own data, but not to those present on the other nodes. The information exchanged between nodes can be model parameters, gradients, or other model-specific information.

Once you choose an AIF distribution strategy, you must provide a series of additional parameters for cases of distributed or federated AIF. These parameters are:

- distributionMode: client-server/peer-to-peer, which describes a client-server or P2P AIF,
- distributionClientNumber: which describes the number of clients that participate in the AIF network,
- minClientAvailable: which describes the minimum number of clients needed to execute AIF model parameters update,
- exchangeChannel: which indicates where the data/parameters are exchanged,
- exchangeChannelProperties: properties to identify needed components.

AI Model Descriptor. With the model descriptor we give some basic metadata information about the model(s) used in the AIF and some i2 interface specifications. A model can be retrainable, declaring this way a dynamic update. This requires the implementation of an interface that triggers the activation of the retraining.

Dependencies. Apart from implicit dependencies, it is possible to define also explicit dependencies of AIF with other functions. This may be necessary, e.g., when the prediction function requires the corresponding continuous training to update.

2.6 Cross-layer, Multi-Connectivity Aggregation and Scheduling Technologies

In deliverable D4.1, we presented the AI@EDGE approach for multi-connectivity and related scheduling challenges. The aggregation of multiple interfaces, 3GPP and non-3GPP ones, is indeed a promising direction to further increase the reliability and the throughput guarantees in environment where multiple radio-access-technologies are made available to UEs and applications.

As anticipated in the 5G system Release 17, this integration is referred to as ATSSS (Access Traffic Steering, Switching and Splitting), making use of Multipath Transmission Control Protocol (MPTCP) with required capability at both the UE and the 5G Core network (5GC). In the AI@EDGE project, we go beyond the ATSSS system and also evaluate alternative approaches where the solution is transparent to the 5GC (we refer to as the off-path mode in D4.1) using an off-path (out of the 5GC access path) proxy, or where the application can be made MPTCP capable (as in the recent months MPTCP was integrated in many Linux releases) hence avoiding the proxy.

Moreover, we also consider the aggregation of interfaces other than WiFi with 5G. We encompass the possible coexistence of 4G, LiFi, and wireline Ethernet with 5G and WiFi, in particular in the frame of the project's Use Case 4 (D5.2), where on-seat screens and bring-your-own-device equipment can make use of these interfaces. In such a heterogeneous environment, efficient scheduling is difficult to achieve.

The goal of T4.3 is in this setting to design a solution that, making use of online analysis of metrics reports from the different UE access interfaces, is able to predict where packet loss will occur and therefore adapt the packet streams sent in the downlink interfaces via the different access technologies. A predictive scheduler has been designed and integrated into the MPTCP-Proxy in off-path mode. The research space in the predictive scheduler design is already presented in D4.1.

2.6.1 Integration of MPTCP

In this section, we describe the integration of MPTCP-proxy in AI@EDGE Testbed, which is further elaborated in Section 3. We used srsRAN version 21.10 [6] to deploy UE and gNB. The Radio connection between the two devices is established by X310 SDRs. For the WiFi connection, we use OpenWRT version 19.07 to set up WiFi connection. The MPTCP proxy is hosted inside an Intel NUC, which is connected to Public Data Network through WiFi connections.

In AI@EDGE use-cases, and in particular in UC4, we are investigating usage of MPTCP to increase onboard wireless communication performance. MPTCP makes use of multiple interfaces, for example WiFi

and 5G, to carry single data stream. Within the UC4 of the AI@EDGE, both the onboard clients (e.g., removable display unit, RDU3) and servers (e.g., system control unit, SCU3) can benefit from the capabilities of MPTCP. However, to use an external server which is not MPTCP capable, the integration of an MPTCP-proxy (Figure 16) is required: In this scenario, the proxy converts MPTCP traffic into standard TCP before forwarding it to the none-MPTCP server. Conversely, when the proxy receives TCP traffic from the server, it converts traffic into MPTCP before sending it to the client.

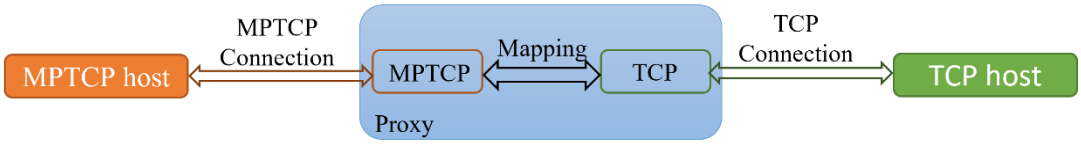


Figure 16 MPTCP proxy

The MPTCP proxy has been deployed to fully benefit from an MPTCP-enabled RDU that can use both WiFi and 5G simultaneously. A studied integration is shown in Figure 17, representing a positioning of the MPTCP proxy "after" the 5GC, in order to support an over-the-top operation independent of the 5GC, differently than the ATSSS.

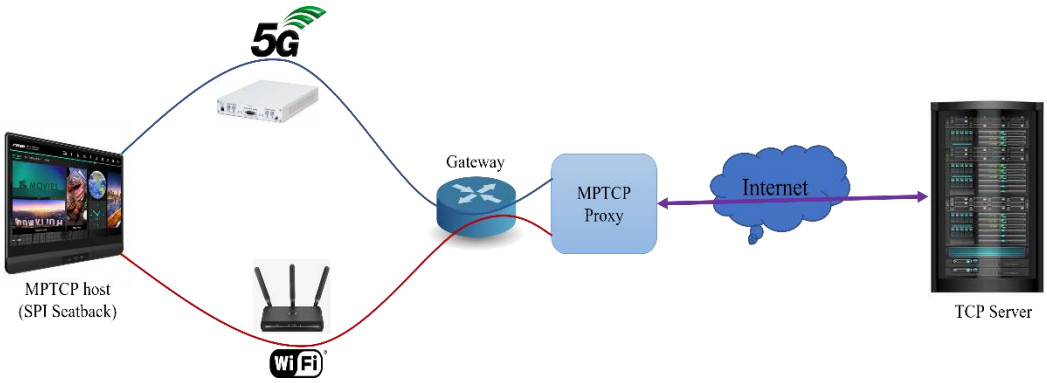


Figure 17 On-path MPTCP model configuration

We have also explored other integration, such as incorporating MPTCP directly at the server side, hence not requiring a proxy.

2.6.2 Predictive scheduler

Figure 18 depicts the predictive scheduler's system architecture. The predictor analyzes metrics reports from the RAT subsystems in order to anticipate packet loss, as well as delay variations, before they are notified (or inferred) via TCP signaling. It then updates the MPTCP proxy/server and in turn, the MPTCP scheduler decides to either change the downlink interface, duplicate the packet, or adapt scheduler interface weights for load-balancing purposes.

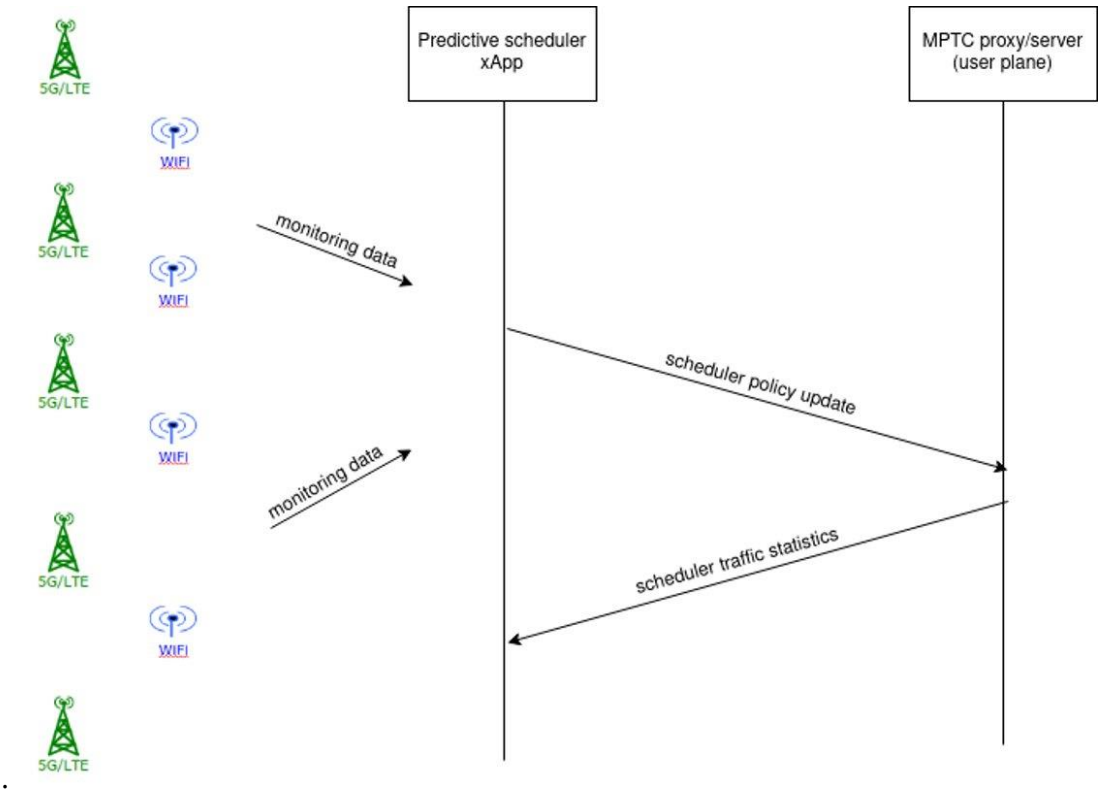


Figure 18 The prediction system's architecture

As a preparatory step towards the actual evaluation phase of CCP, we wanted to determine the most appropriate platform for the xApp development and execution; we did performance tests of the prediction algorithm at the near-RT RIC performed to assess which time complexity can be supported. The autoregressive integrated moving average (ARIMA) algorithm was used to build the predictive model: we tested its application on both C and Python implementations as an application within the 5G-Empower environment. The input data set is collected every 80ms. The prediction algorithm is responsible for predicting future values in two cases: the first one has the same duration as the input and the later has fixed duration for 5s. The advantage of using C-based implementation rather than a Python-based one is measured to be from 10% to 50% in terms of execution time. These preliminary tests allowed us to determine that using system calls from python-developed xApps seem the most time efficient way forward. These tests were run using arbitrary data. Then, we tested the ability to predict packet loss based on metrics collected from gNB. With that purpose, we built an ad-hoc testbed which uses a modified version of srsRAN to be able to record RAN metrics. Thanks to srsRAN channel emulation function, we could set up RAN in 4 different scenarios:

- Normal with perfect RAN channel.
- Fading channel: Extended Typical Urban (ETU300) model with a maximum doppler dispersion of 300 Hz.

- High Speed Train Doppler model simulator: High speed train scenario described in 3GPP 36.101 Section B.3.
- Radio-Link Failure.

RAN metrics were collected at two different rates of 100ms and 1s, each over a 10-hour period (we call this one is data 1). Along with collecting the metrics, we also recorded all the packets exchanged between the UE and the server in the form of PCAP (we call data 2). The analysis of data 2 allows us to determine when the packet loss occurred, connecting to data 1 we build a new data which contains the information of the RAN metric and the corresponding packet loss event. This data is used with the anomaly detection LSTM model with 2 hidden layers of 64 neurons to obtain a packet loss prediction model. The resulting model is used in a prediction of packet loss in the next step. Figure 19 shows the time required to collect data from srsRAN gNB and the time required to get a prediction result upon request.

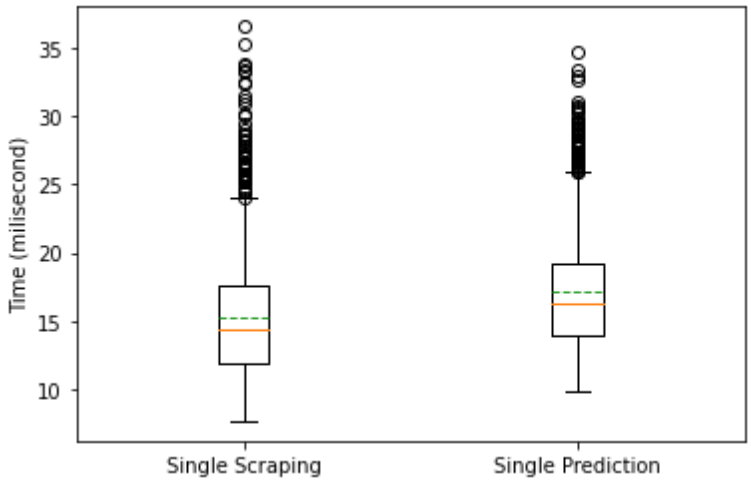


Figure 19 Profiling of scraping and prediction time

We assess the model's performance using Precision and Recall metrics alongside Accuracy. This choice stems from the nature of our prediction problem, which involves unbalanced binary classification. In this scenario, instances of packet loss (value 1) are significantly less frequent than instances where packet loss does not occur (value 0). Therefore, relying solely on the Accuracy metric for model validation is insufficient, as it would consistently yield high values. Here are the detailed definitions of these metrics:

- Precision measures the percentage of correctly classified positive predictions among all positive predictions.
- Recall measures the percentage of correctly classified positive predictions among all actual positives.
- Accuracy represents the percentage of correctly classified observations.

Frequency	100 ms		1 s
Sequence length	70		7
Time step	t+1	t+10	t+1
Accuracy	98.97%	97.88%	93.78%
Precision	90.49%	71.01%	73.16%
Recall	67.73%	31.95%	50%

Table 1 Performance of LSTM model

Table 1 presents the evaluation metrics for our collected dataset at two different sampling rates: 100 ms and 1 s. For the 100 ms dataset, predictions are better at t+1 (100 ms ahead) compared to t+10 (1 s ahead), whereas for the 1 s dataset, predictions at t+1 (1 s ahead) outperform t+10 (1 s ahead) of the 100 ms dataset. However, we favour the 100 ms dataset because it allows us to make earlier predictions compared to the 1 s dataset. Additionally, we evaluate these metrics at multiple time steps ahead, not just at t+1 (100 ms ahead). In Figure 1, we observe a decreasing trend in model performance (Precision and Recall) as the number of time steps increases.

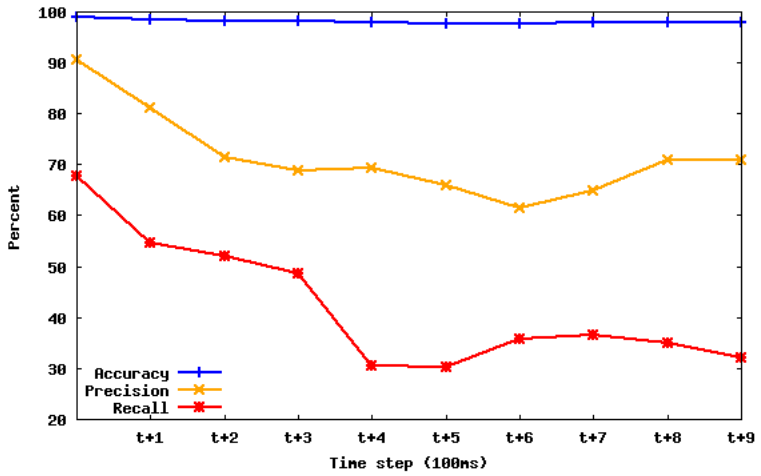


Figure 20 Performance of 100ms dataset at multiple time steps ahead

In the next step, we deploy predictors as xAPPs within a near-real-time RIC environment. These xAPPs utilize the previously created predictive model to forecast packet loss. Subsequently, the editor updates the MPTCP proxy/server, leading the MPTCP scheduler to determine actions such as altering the downlink interface, duplicating packets, or adjusting scheduler interface weights for load balancing. The E2 interface, detailed in section 2.3.2, enables the xAPP in the near-real-time RIC to gather data from the RAN segment. This information serves as input data for predicting packet loss using the model.

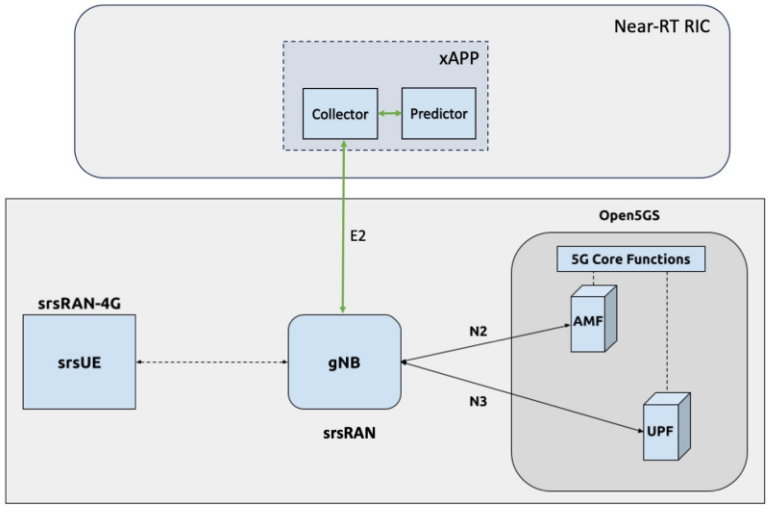


Figure 21 xAPP testbed

To test the model, we deploy a testbed using flexRIC as a near-RT RIC environment, alongside an xAPP equipped with the initial version of the E2 interface on srsRAN gNB, as shown in Figure 21. The xAPP establishes a connection with the srs gNB agent (more details are provided in section 2.3.3) and collects data at 150ms intervals. It's essential to acknowledge that there are limitations in the open-source code of the open-RAN srs gNB and the near-real-time RIC environment. Currently, the srs gNB can collect only three metrics, while the flexRIC xAPP can collect just one. Nonetheless, testing the testbed's operation within these current limits demonstrates the feasibility of our proposed model. Furthermore, it highlights the model's flexibility for future extensions once full parameter support is implemented, taking us closer to a comprehensive solution.

After the predictor forecasts potential packet loss on a downlink interface, the information must be conveyed to the MPTCP scheduler to make the appropriate decision. In our scenario, this information is sent to the MPTCP-proxy. A process running in user space will be responsible for receiving this information and communicating it with the MPTCP scheduler.

In MPTCP, the scheduler operates in the kernel and is responsible for distributing traffic across available paths based on a specific scheduling policy. More precisely, for each MSS (Maximum Segment Size) of data, the MPTCP scheduler selects one or more available paths for transmission. The default scheduling policy in MPTCP is based on round-trip time (RTT) estimates, favouring the path with the smallest RTT. Selection of the MPTCP scheduler can be done at the user-space level for the entire system or for each socket via parameters during socket initialization. However, changing the scheduler policy requires recompiling the kernel. Some recent proposals enable the use of eBPF to modify the scheduler policy without requiring kernel recompilation. However, these new solutions are limited to basic scheduler policies and do not yet support changing the scheduler policy for an existing connection.

Additionally, MPTCP supports netlink, a communication protocol used in the Linux kernel for communication between kernel space and user space. It is utilized to enable intervention in the process of establishing paths between sender and receiver when initiating an MPTCP connection. However, using netlink introduces some latency, making it suitable primarily for the initial connection initialization phase of MPTCP.

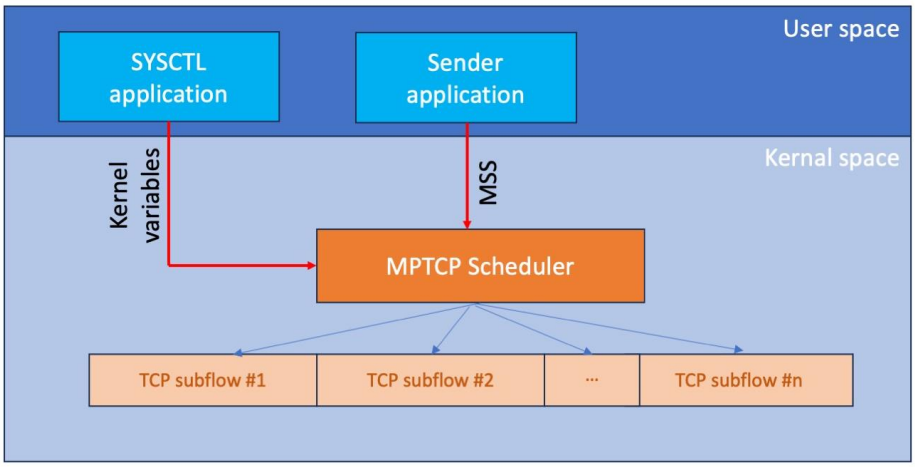


Figure 22 Integration of packet loss prediction into the MPTCP scheduler

To minimize changes to the source code of the MPTCP scheduler, we employ a straightforward solution to demonstrate our model. We pre-declare certain kernel variables that can be modified via the sysctl tool from user space (see Figure 22). The MPTCP scheduler utilizes these kernel variables as parameters when comparing the RTT of different paths. Therefore, the MPTCP scheduler temporarily avoids using path where it is predicted that packet loss may occur.

2.6.3 Emulation for the creation of datasets

Waiting for the full availability of datasets from real end-to-end testbeds, and to allow large-scale evaluations, we worked toward the creation of datasets to use for experiments building on an emulation testbed, emulating the RAN segment. We set up a simulation/emulation hybrid system, using a combination of NS3 (The NS3 network simulator, n.d.) network simulation tools with a realistic virtualized network system software stack. Using anonymous user traffic data from a national mobile network provider, we employ NS3 to simulate real-time user (client) data, which is then transferred from the simulated environment to the real network. **Error! Reference source not found.** depicts the client-server view of the d ata-plane traffic flow over the platform, where we simulate the application connection handling on both the User Equipment (UE) and the server sides. Each UE has one NS3 instance that generates the application traffic for each TCP session from the mobile traffic dataset.

IP traffic from NS3 node is transferred into the UE's network to go through the mobile core network and the WiFi backhaul network, concurrently, and then to the end-server. On the server site, network traffic is transferred back into NS3 to reach the application server emulated in NS3. Similar behaviour happens when traffic is sent back from the server application. During these data transfers, we collect information on the

overall network system components (including both 5G NSA and WiFi access points) at different sampling rates, thereby building a dataset composed of thousands of time-series, each related to a specific subsystem and a specific resource type.

In NS3 we use a simulated topology that includes multiple application nodes connected to one gateway. Each node is responsible for simulating the traffic of one TCP session in real-time. On the server side, multiple NS3 instances are used, each instance being responsible for simulating a set of application servers. Also, a virtual tunnel is created to allow traffic from NS3 to be sent/received to/from the UE's network. In order to create sets of connections to emulate the real communications, NS3 is carefully configured based on the input dataset.

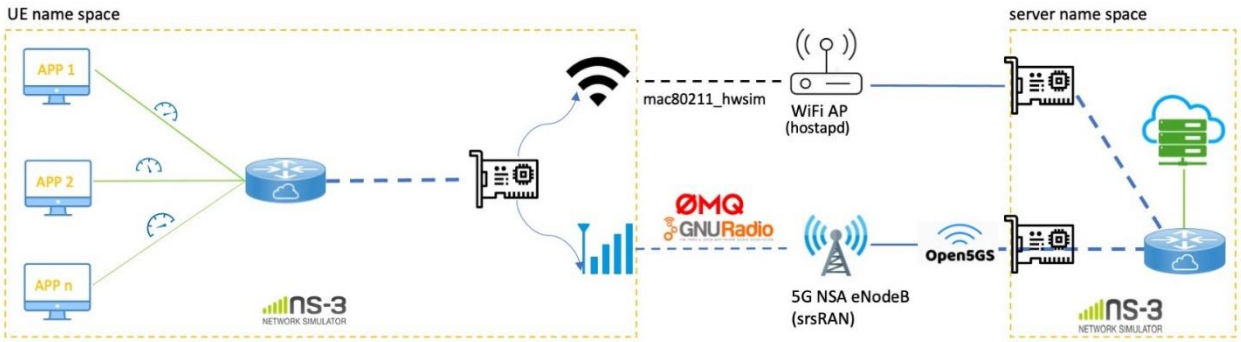


Figure 23 Hybrid emulation platform for stack monitoring dataset generation: Per connection view

For the RAN part, we use srsRAN version 21.10 [6], a 4G/5G software radio suite developed by SRS (Software Radio Systems), to deploy UE and gNB. The radio connection between UE and gNB is done via GNU Radio, an open-source tool that provides signal processing blocks to implement software-defined radios and signal-processing systems.

To emulate WiFi connection for UE, we use mac80211_hwsim, a Linux kernel module that can be used to simulate arbitrary number of IEEE 802.11 radios. It works by tracking the current channel of each virtual radio and copying all transmitted frames to all other radios that are currently enabled and on the same channel as the transmitting radio. We use Hostapd, a user space daemon that acts as an Access Point (AP) and authentication server, to emulate the AP node.

The core of the 5G network is deployed through the Open5GS software suites [7] version 2.4.4, related to 5G Non-Stand-Alone (NSA) system; in future releases of the dataset, we plan to move to a 5G SA system. After their containerization, we use Kubernetes [8], to deploy these basic core functions used in the NSA Open5GS version: HSS, MME, UPF, PCRF, NRF, SGW-C, SGW-U, SMF on three physical servers.

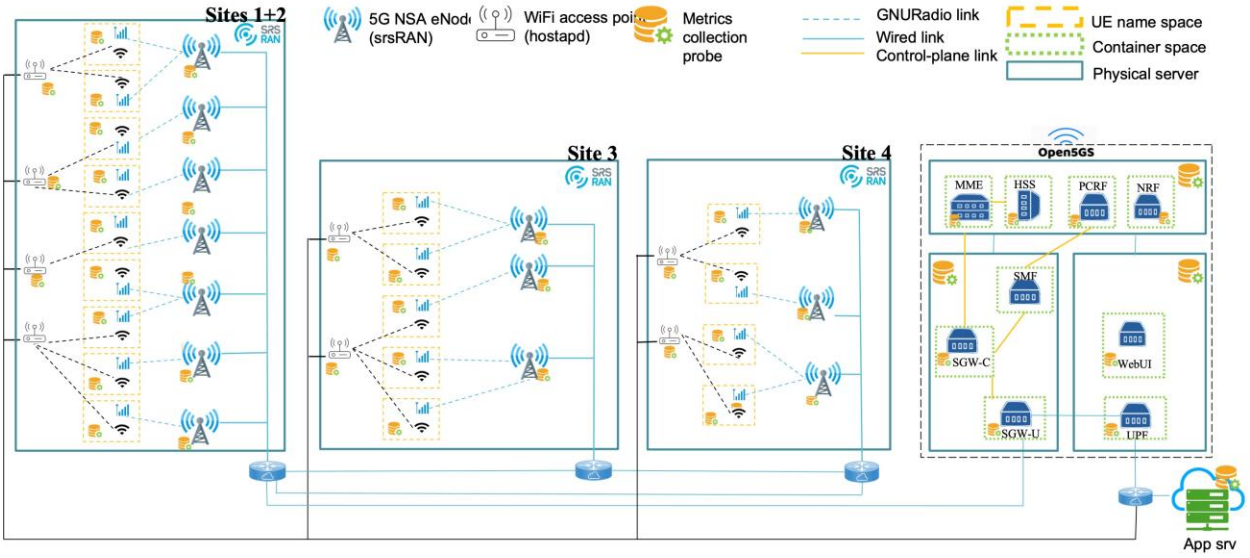


Figure 24 Hybrid emulation platform for stack monitoring dataset generation: topology

From the mobile operator network, we select a small cellular region (TAC, Tracking Area Code). The whole area is covered by 4 towers-sites with 13 cells; the number of users with TCP sessions over 100~kbytes counted up to 52 UEs. Figure 24 depicts the physical model of the system set-up to emulate the scenario. Three powerful servers connected in a triangle are used to simulate 4 sites. Sites 1 and 2 have a total of 7 cells, while sites 3 and 4 have 3 cells. Each cell serves a maximum of 5 UEs (the figure does not include all UE due to constraints in available space). Based on the information about cell locations at the start of the session, we assign a corresponding set of TCP sessions to each site. Note that the 5GC is deployed on a clustered system with 3 physical servers, one represents the master node, and the three others are used to deploy components of the core network. One physical server is used to simulate the server site; to avoid congestion, the server is not shared between UEs, and each UE has an independent set of servers for download and upload.

In order to create a first dataset that can be used for training purposes, i.e. to learn the network normal conditions in work package 3, the system ran for 14 days, utilizing the mobile traffic traces described above These data can be considered as data representing the normal operation of the system. In the second step, we created a test set with the objective to test anomaly detection algorithms such as the one in [9]. We inject some anomalies into the system, while gathering metrics. Currently, we focus on 4 injected anomaly behaviors: CPU overload, Access bandwidth, Packet loss and Link failure.

2.7 Hardware Acceleration Solutions for AI/ML

The introduction of HW acceleration in the CCP is crucial for the success of AI/ML execution in real-time MEC application/scenarios. In this direction, the challenges are to provide seamless integration of multiple diverse HW devices (GPU, FPGA, etc.) in the platform and the MEC scenarios of AI@EDGE, as well as to facilitate easy development of SW accelerated kernels with substantial benefits in terms of execution time and/or energy. An initial development analysis was presented in deliverable D4.1, Section 7 [1], which constitutes the baseline solution of AI@EDGE. Additional developments towards the final solution are presented here in subsections including detailed descriptions for each one of the new developments.

2.7.1 Installation of HW accelerators

As described in D4.1, the acceleration-enabled cluster being developed in ICCS premises comprises of four Kernel-based Virtual Machines, deployed on top of three different Intel Xeon Physical Machines to provide heterogeneity both on CPU-family and compute-resources level. Those Physical Machines and their characteristics are described in detail in D4.1. On top of the Virtual Machines, we have deployed single-master Kubernetes v1.20.5, the most widely adopted container orchestrator. Virtual Machines have been a design choice since they allow for multi-tenancy at the physical-node level. Therefore, since AI@EDGE leverages cloud-native technologies to enable seamless deployment across the Edge-Cloud continuum, the combination of VMs with containers is currently the common way of deploying clusters at scale, since it establishes the perfect catalyst for reliability and robustness.

In the acceleration-enabled cluster, we assume both near- and far-edge sites. Thus, when selecting the devices to provide hardware acceleration, we need to address both of those use-cases, and to take into consideration their features, constraints, and characteristics, in terms of computing capabilities, power consumption and costs. We used a combination of GPUs and FPGAs with diverse characteristics to provide a variety of choices in this direction: **1 Nvidia GPU Tesla V100** (near edge device) and **1 Jetson AGX** and **1 Jetson Nano** (far edge devices), as well as **2 Xilinx FPGAs Alveo U280+U200** (near edge devices) and **1 Versal VCK190** and **2 Zynq ZCU104** (far edge devices).

The more powerful hardware accelerators were connected via PCI in the physical machines and were made available in the Virtual Machines using the PCI passthrough. In addition, the device plugins [10] provided by NVIDIA [11] and Xilinx [12] allow the addition of custom resources, i.e., GPU, and FPGA to the Kubernetes API, without configuring and redeploying the source code. As a result, the Kubernetes scheduler is aware of the presence and availability of those resources per node and is able to prevent the placement of a pod in an incompatible node that does not meet the criteria (specific accelerator requested is not present or currently unavailable), that otherwise would fail to execute.

2.7.2 Accelerated AIF Development & Containerization (examples in CCP)

The SW toolflows used for accelerating AIFs, including their selection rationale and required coding steps, are described in D4.1, section 7 [1]. The following paragraphs summarize practical implementations for specific examples tested in AI@EDGE WP4: 1) CNN with Vitis-AI, 2) LSTMs with Vivado-HLS, 3)

containerization and deployment of accelerated AIFs. These implementations have led to certain conclusions and guidelines.

CNN with Vitis-AI

Xilinx Vitis-AI is a solution for AI inference on Xilinx hardware platforms, including both edge and cloud FPGAs. Its development environment offers an end-to-end implementation, requiring solely a trained Tensorflow or Pytorch model as input.

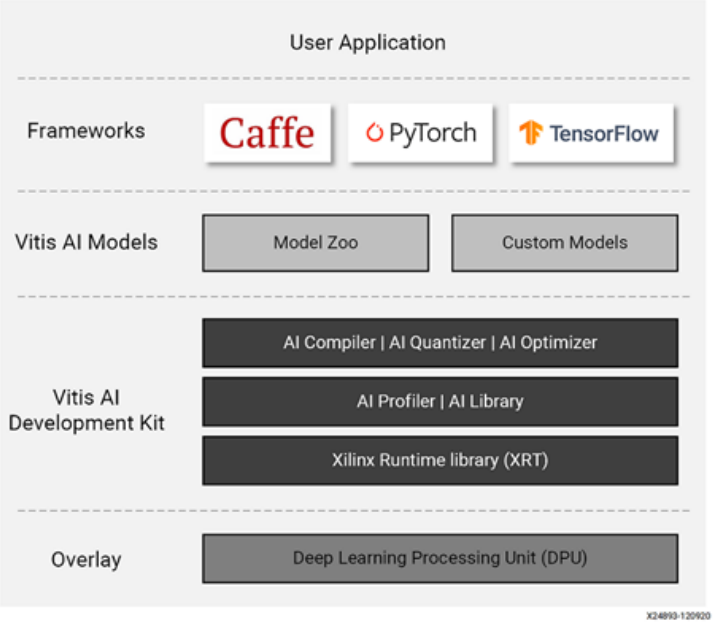


Figure 25 Overview of Xilinx Vitis-AI and DPU tools

It achieves such a feat by making use of multiple optimized 3rdparty functions/cores, tools, and libraries. The highlight of the tool is the deep-learning processor unit (DPU). The DPU is a programmable soft-core implemented in the Programmable-Logic (PL) region of the hardware devices and is optimized for convolution based deep neural networks. Vitis-AI offers a series of different DPUs for embedded devices, Versal ACAPs and Alveo cards, each optimized for the specific device and focused on either power, throughput, or latency.

The development process begins with quantizing the trained model, converting it from a floating point to an 8-bit integer one. Afterwards, the Vitis-AI compiler maps the AI model to a highly efficient instruction set and dataflow model suitable to be used by the corresponding DPU. The end product is a .xmodel file containing all the information necessary for the DPU to run the model. The final step is developing the code that runs on either the server CPU or the embedded processor, using the Vitis-AI Runtime Library (VART) C++ or Python API.

We explored the Vitis-AI functionality by accelerating a variety of convolution based deep learning models on a ZCU104 MPSoC and on a U280 Alveo card, using models trained on Tensorflow2 on Vitis-AI 1.4.1. During this testing, we encountered many hiccups and obtained useful experience enabling us to pinpoint

the limitations of the tool. A list with the restrictions of each DPU can be found on the Vitis-AI Userguide. Some of the issues that arose regarding the architecture of the models are:

1. BatchNorm layers that are not fused with previous layers, are mapped as depthwise convolution with kernel size 1. Those cannot be implemented on the DPU used on the Alveo U280. This issue was spotted on ResNetV2 architectures.
2. Global Average/Max Pooling maximum size is that of 8. This would mean that the ResNet-152 architecture, where the final Global Average Pooling layer is of size 10, could not be accelerated by Vitis-AI. However, by replacing that layer with two consecutive Average Pooling layers with pooling sizes of 2 and 5, the model can be implemented in Vitis-AI. This adjustment needs to be made by hand, as the compiler is unable to automatically make this layer replacement.
3. There are certain size constraints on the Convolutional Operation layers, in terms of channels. During our experimentation, those were exceeded when big Fully Connected Layers were mapped to Convolution layers.
4. The models should contain exclusively ReLU or ReLU6 activation functions when it comes to U280 implementations and ReLU, ReLU6, LeakyReLU, Hard-Swish or Hard-Sigmoid when it comes to MPSoC implementations.

On top of the guidelines concerning the choice of models, there are instructions focused on the Tensorflow implementation of the models. The Tensorflow2-specific guidelines are:

1. Use the Functional API with standard layers.
2. Avoid the use of custom layer subclassing.
3. Avoid the use of models as layers as part of the functional API
4. Use Tensorflow 2.3 Version, models trained on Tensorflow 2.6 were bugged when loaded to Tensorflow 2.3 environment.

LSTM in Vivado HLS

We used High-Level-Synthesis (HLS) to implement the accelerator kernels of the LSTM models. The device setup was a Xilinx's U280 FPGA for near-edge nodes and Xilinx's Zynq ZCU104 MPSoC for far-edge nodes. By using HLS along with C-code instead of a low-level description language we achieved faster development and design space exploration. The acceleration flow was done using directives or "pragmas" in the C-code that guided the HLS compiler to synthesize the kernels into FPGA hardware. During our hardware design we assumed various LSTM sizes and performed a design space exploration to determine the most efficient acceleration approach. The computationally intensive parts of the LSTM algorithm, in terms of required processing time and number of operations (determined by the number of units and features of the LSTM cell), are the calculations of the four activation vectors (i_t, f_t, $_t$, o_t), as explained in [30]. Parallelization at those calculations is necessary, especially for a low latency implementation. Next, we present a summary of the considered acceleration techniques.

- Parallelization at the layer level, where multiple layers would process data in parallel. This parallelization technique could achieve a speed up of maximum the number of layers. Disadvantages in this implementation include increased resources, underutilized hardware and extra logic for synchronizing and buffering, thus it was not an ideal choice for acceleration.
- Parallelization inside each LSTM cell at the iteration level, where multiple time steps, for different input data could be processed in parallel using a systolic array architecture. Some of the disadvantages using this technique were increased resources, less scalable architecture, and an implementation that does not reduces the latency.
- Parallelization inside each LSTM cell at the calculation of the activation vectors. Considering the resources of the targeted FPGA, the developer can choose to process in parallel one or more of the activation vectors and also parallelize each calculation internally. This was done using a combination of pipeline pragmas to pipeline the loops and unroll pragmas to create multiple independent operations of the inner loops and parallelize the execution.
- Moving to a fixed-point architecture instead of floating-point can reduce further the latency. This change reduced the required resources of the kernel and the memory bandwidth requirements.
- For speeding up the communication between the kernel and the DDR memories of the device, a wide memory interface was utilized by using the HLS data type interface ap_uint<512> for taking advantage of the 512-bit bus of the device.

Our main architecture, presented in Figure 26 has four parallel engines, one for each activation vector in the LSTM and inside each of those engines we parallelized the multiply-accumulate operations for each row of the matrix with the active vector. To achieve the desirable parallelization the active vector and the weight matrices were partitioned in their entirety and second dimension respectively, using the HLS partition pragmas. The HLS kernel is written in a way that can easily be modified to support different number of layers, units and feature values, and parallelization factor. The number of time-steps is received as an input value by the kernel.

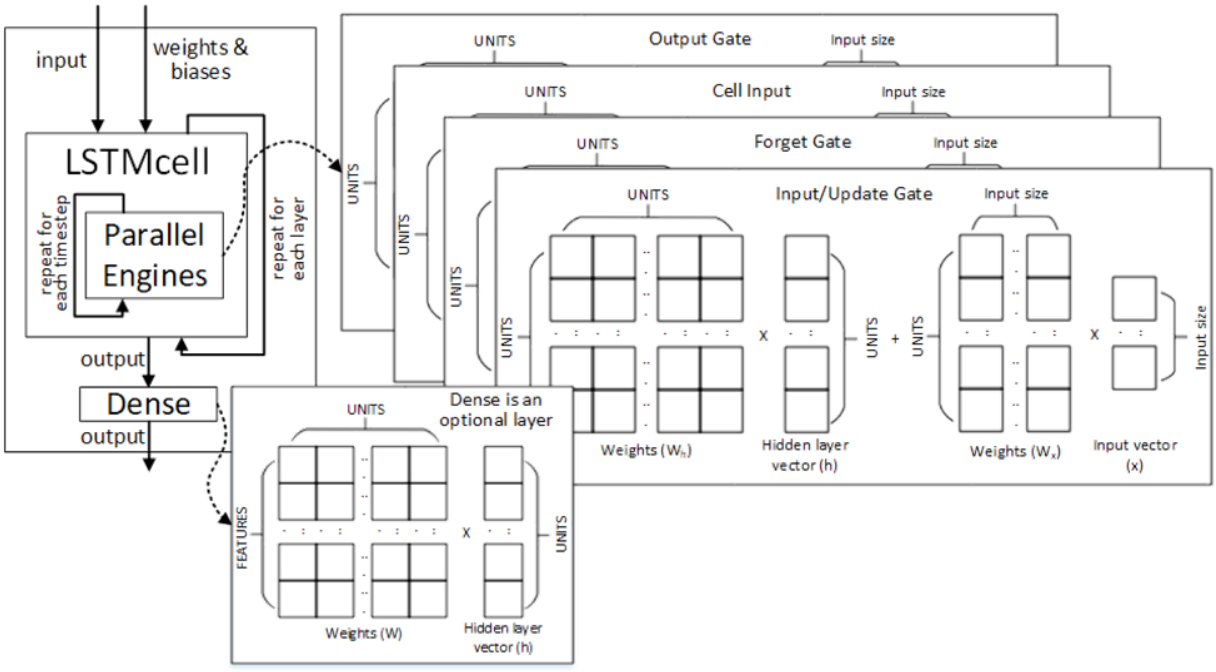


Figure 26 Parallelization of LSTM kernels for FPGA acceleration

AIF containerization and deployment steps on the Connect-Compute Platform

To include the accelerated AIFs in the CCP, we need to put them in containers, similarly to all SW-based AIFs. Using the Vitis-AI base docker-image, we have implemented a ResNet50 model, trained on ImageNet, on Alveo U280. This AIF utilizes the hardware resources in the hardware accelerator via the Vitis-AI framework, as described above. In order to avoid re-initializing the kernel of the card, we have implemented a Python code wrapper, that initializes the kernel once, and implements an API that allows for external clients to access the service. The endpoint becomes available outside the cluster by exposing the service through the hosting node. We build the final docker image, and differentiate it from other versions (CPU-only, GPU), using a device- and implementation-specific tag, i.e., cnn:alveo-u280-latency.

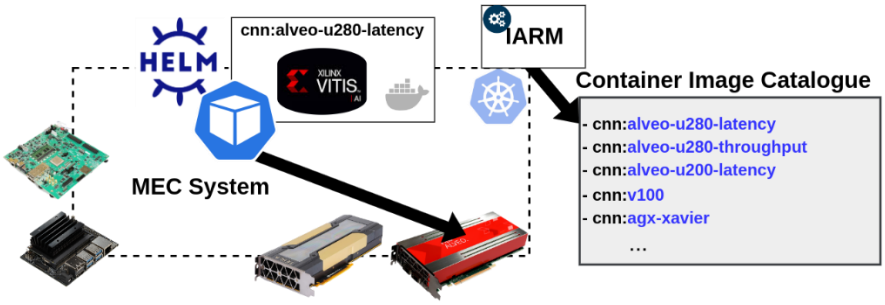


Figure 27 Deployment of accelerated containers

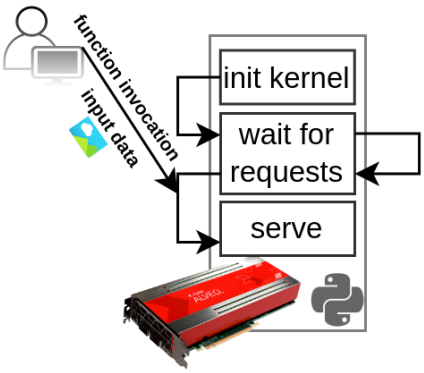


Figure 28 States of accelerated AIFs

The image is then pushed into the AI@EDGE AIF catalog, where all the available AIFs, as well as the different versions of the same AIF are stored. For AIF deployment, users utilize the AIF descriptors of AI@EDGE. In case an Acceleration-Enabled MEC System is selected for the deployment of the incoming AIF, the MEO, except for the IOC, will employ the Intelligent Acceleration Resource Management (IARM), an Intelligent component, designed to be used on MEC Systems that provide hardware acceleration capabilities. IARM, as was mentioned before, based on the AIF description (available versions of the incoming AIF, user requirements, and optimization goals), and the current state of the system will prioritize the available nodes/devices. The resulting output will be forwarded to IOC or MEO via HTTP to make the remaining actions for the deployment of the AIF through the MEC Platform Manager (PM).

Following the cloud-native deployment paradigm of the project, AIFs are containerized applications, that are managed by Kubernetes, MEO and MEC platform. The minimum unit of deployment in Kubernetes is the pod, which includes one or more containers, which share the same (virtual) IP address, network ports and storage volumes. Pods are usually defined in a YAML configuration file (or via the Kubernetes-provided API). The configurations parameters include labels, mounted volumes, exported ports, pod replicas (in a Deployment config file) and others.

Additionally, as Figure 29 depicts, in order to inform Kubernetes scheduler about the need of specific resources, we need to request the respective resources, i.e., xilinx-alveo-u280_<xdma-version>. Afterwards, Kubernetes will verify the node-feasibility (existence and the availability) regarding the

requested resource and apply a first-level filtering of the nodes. In case this resource is available, kube-scheduler will select the most viable node according to its priority functions to place the incoming pod. In our case, Kubernetes will skip its internal scheduling process, to adhere to the indications provided by the Intelligent components, e.g., IOC, IARM, implemented in AI@EDGE.

```
kind: Deployment
metadata:
  name: cnn-u280l
spec:
  selector:
    matchLabels:
      app: cnn-u280l
  replicas: 1
…
      containers:
      - name: cnn-u280l
        image: <username>/cnn-test:u280l
        imagePullPolicy: Always
        volumeMounts:
        - mountPath: /opt/xilinx/overlaybins
          name: overlaybins
…
        resources:
          limits:
            xilinx.com/fpga-xilinx_u280_xdma_201920_3-1579649056: 1
        ports:
        - containerPort: 3000
```

Figure 29 Acceleration-enabled AIF deployment YAML configuration file

2.7.3 Framework for Automated Code-Image Generation of AIFs in Heterogeneous Clusters

The HW heterogeneity of clusters consisting of multiple kinds of nodes and accelerators poses a challenge on seamless deployment of AIF SW anywhere/anytime in the CCP. As explained above, every different FPGA, GPU, x86, or ARM architecture requires its own executable. To facilitate the deployment to CCP of an AIF provided by a user at high-level language description, e.g., Python Tensorflow, we develop an automated framework that inputs the provided AIF code and generates multiple distinct container images, each one containing executables+libraries suitable for a specific HW node/accelerator. Given the programming context selected for AI@EDGE, i.e., high-level programming and acceleration tools instead of low-level coding/optimization, this proposed framework is realized via meticulous integration of tools and describing multiple HW cases.

More specifically, our framework is developed in Python and requires the following inputs:

- An AI model created in the Tensorflow2 framework.
- Some server-side code to specify the input decoding, data preprocessing, data postprocessing, output encoding and other experiment-specific code.
- Some configuration files that specify AIF details (Batch size, server IP etc).

- A dataset for the client to send to the server (optionally).
- Some client-side code to specify the data transfer between client and server.

Based on these, our framework converts the Tensorflow2 model to the appropriate form to be used by the different AI-inference frameworks we use for each HW/SW platform of the CCP (each node/accelerator). Figure 30 summarizes the integrated tools: we use the Tensorflow Lite framework for CPU and ARM platforms, we use Tensorflow-TensorRT for GPU platforms, Xilinx Vitis-AI for FPGAs and ONNX Runtime with TensorRT backend for Xavier AGX. Afterwards, it combines the user provided code with the automatically generated code, which considers the intricacies and restrictions of each vendor's framework, to realize a server environment for each HW/SW platform. Finally, based on the user provided configurations, it creates a working AIF container for each platform and uploads them to a repository, ready for deployment to CCP. One additional feature is the creation of an example client container that is designed to appropriately interact with the AIF servers/containers.

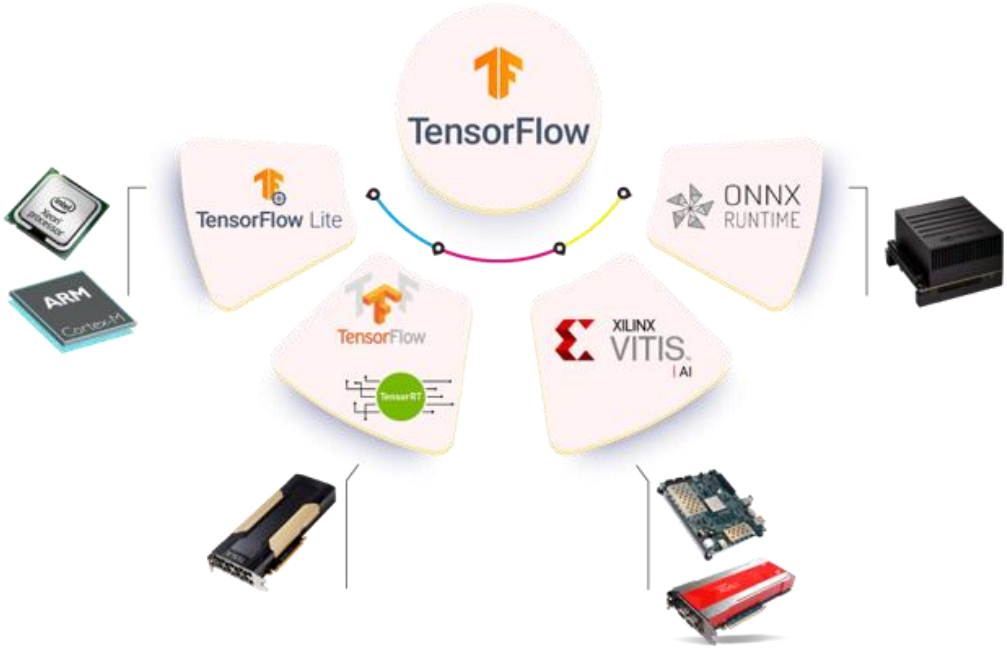


Figure 30 Tools integrated in the proposed Code-Image Generation Framework

Furthermore, mainly for testing purposes, we also automate an AIF-level metric collection. We implement a small Python tool which, given the platform specific containers of an AIF, implements multiple runs and keeps records of the performance. The metric collector aggregates the system metrics of the platform before and during the AIF run, as well as the AIF-level metrics, provided by the Accelerators Monitoring Framework (see below), for each AIF run. After running the AIF multiple times, it collects the required AIF-level metrics from the in-memory intermediate storage of the cluster, that are specific to that set of

AIF-runs. Based on these, we can create a dataset consisting of the system metrics and the AIF-level metrics of a specific AIF-run, which can be used for AI training of resource managers.

2.7.4 Intelligent Acceleration Resources Manager

In the CCP, the MEC Systems with hardware acceleration enabled nodes will include a customized component for Intelligent Acceleration Resource Management (IARM). The IARM cooperates with IOC and MEO, at higher-level, to provide the additional functionality required for leveraging the hardware accelerators' capabilities in a multi-tenant environment with various AIFs arriving dynamically. IARM provides an API endpoint that allows other components, i.e., MEO, to relay the AIF description to it. Depending on i) the user-defined optimization goals, ii) the available versions of the AIF, e.g., images for GPUs or FPGAs, iii) the current HW availability, and the current state of each node that includes a hardware accelerator, and iv) the accumulated knowledge of the IARM, acquired from previous runs and pre-profiling, the most feasible node will be selected and returned as a response to MEO.

Depending on MEC system policies, the optimization goal can include latency, throughput, or energy minimization, while meeting other constraints, e.g., cost. Regarding the available versions/executables of the AIF (each of them implementing the same functionality but with different acceleration), users that construct the AIF description will be able to list the image versions of their preferences (available in the AIF catalogue). Therefore, the platform provider managed IARM, taking into consideration the available images, will make the most viable deployment decision to satisfy users' requirements and utilize, at the same time, the available system resources in an efficient manner, e.g., fair scheduling. Secondly, IARM will be aware of the system- and application-level metrics. Those metrics will be exported from cluster devices and applications and sent to the monitored metrics aggregation system, i.e., Prometheus, of the MEC System. IARM will rely on internal knowledge/rules to derive the best mapping of AIFs to Devices. The knowledge/rules can be updated by collecting information from offline and previous AIF executions.

2.7.4.1 A Proof-of-Concept Design of IARM

The aforementioned functionality of IARM can be realized in a multitude of ways in the context of a very active research field regarding resource management. In the context of AI@EDGE, we design a generic IARM as a proof-of-concept module for completing the structure of CCP. We develop its basic functionality to accommodate the actual orchestration procedures of CCP while, in parallel, we do a first iteration of research towards improving IARM's performance. In the following paragraph, we propose the key components/algorithms of IARM, which can be individually enhanced in the future by performing more in-depth research per component; on one hand, the proposed structure allows us to integrate CCP and demonstrate basic functionality, while on the other hand, it allows us to collect data from CCP and perform research offline towards improving IARM independently of the integration work.

The proposed IARM design is inspired by and combines ideas from three major subfields of Artificial Intelligence: deep-learning AI, classical ML, and expert systems. More specifically, Figure 31 shows the block diagram of IARM, with Reinforcement Learning (deep AI), clustering/classification (classic ML), and simple rules (expert system) for making the final decisions regarding the assignment of an incoming AIF to a HW node. The former algorithm, RL, will handle dynamically the scaling of AIFs in terms of CPU

cores. For example, when an already active AIF is running low in QoS, the RL will propose an action such as scaling of cores, migration to another CPU node, or even migration to an accelerator of the cluster. At the same time, the RL component will learn how to avoid creating interference with its decisions and hindering the performance of the accelerators in the cluster. Its learning procedure can take place even during the lifetime of CCP, online, with the expert system filtering out proposals that are obviously wrong (e.g., downscaling of an already struggling AIF). The ML component will continuously record system and AIF metrics during the lifetime of CCP, such that executing clustering/classification algorithms on these data will expose statistically beneficial assignments of AIFs to HW nodes. The ML algorithm can be executed either offline, statically, or online with more automated procedures in place. In the form of a Look-Up-Table, these SW-HW assignments/statistics can be then passed to the expert system, which will utilize them for deciding the initial placement of an incoming AIF to a node/accelerator. ML can also be employed to match the characteristics of the incoming AIF to the characteristics of past AIFs in order to further accommodate the decision of the expert system. The latter component will rely on a set of simple rules to make final recommendations to MEO or generate alarms (e.g., when detecting high cluster utilization or impossible new assignments to certain accelerators). The proposed structure allows us to develop a preliminary version of IARM that can be easily integrated to CCP, i.e., including a small set of rules and a small pre-calculated Look-Up-Table, while at the same time allows us to divide the RL and ML tasks to facilitate further research.

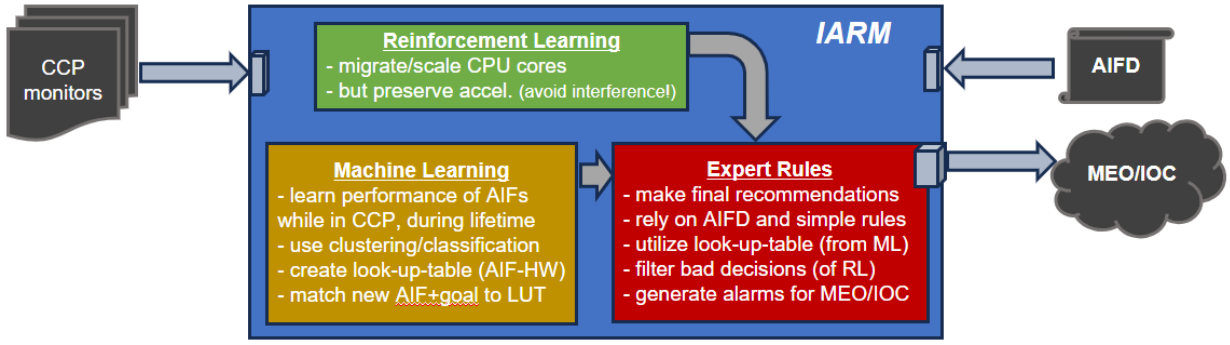


Figure 31 A proof-of-concept structure of IARM enabling integration to CCP, in practice, as well as more in-depth, offline research per component

The above-described operation of IARM relies on the input of AIF descriptors and CCP monitors. The former will allow IARM components to know the nature and requirements of each incoming AIF, and hence, be able to propose assignments and resource management, dynamically. It is for such purposes that the AIFD has been designed to also include this kind of information (see Annex). The monitoring of CCP metrics, such as system utilization and AIF performance is critical for the operation of the RL and ML components of IARM; the current state of CCP is important for knowing HW availability and for estimating the performance of SW when placed on certain HW, while the current performance of SW is important for proposing a migration when necessary. Moreover, the time granularity at which IARM inputs such information is critical for making timely decisions. Therefore, we develop our own monitoring subsystem to accommodate resource management in CCP.

2.7.4.2 Monitoring Scheme for IARM

To accommodate the decisions made by IARM, we need a scheme to constantly monitor the behavior of CCP infrastructure and AIFs. We develop monitoring mechanisms for two distinct types of information: for system- and AIF-level metrics. The former mechanism tracks the nodes' health, availability, power consumption, temperature, or any other kind of feedback from the infrastructure of our system. The latter mechanism handles application-specific metrics, e.g., AIF latency or throughput, the nature of which can vary depending on the user. Both mechanisms are specific data collectors in the NSAP data collecting pipeline architecture described in project Deliverable 3.2 [3]. The monitoring scheme of AI@EDGE is multi-tiered spanning all across the continuum, from the distributed nodes in the edge of the network, to the centralized, Cloud-based NSAP-level. The entire monitoring-related framework is packaged in a helm chart for seamless deployment on a per-node basis.

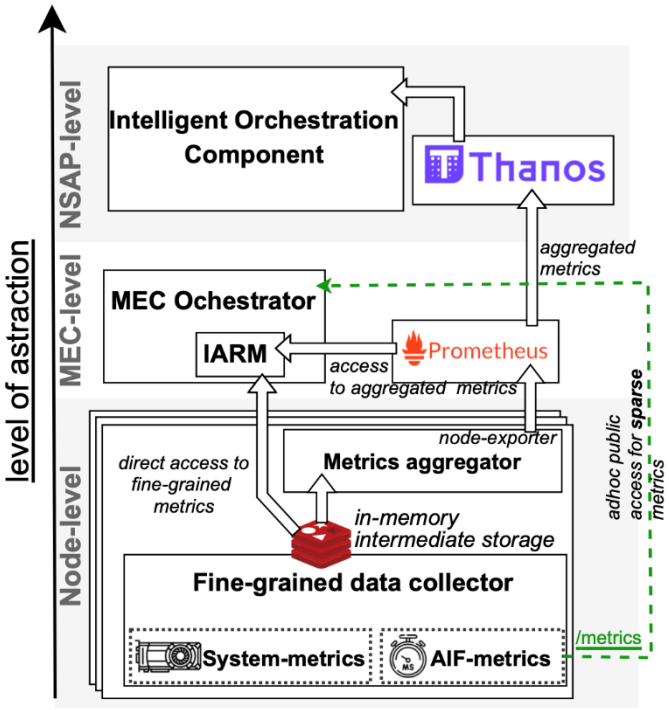


Figure 32 The proposed monitoring scheme for Intelligent Resources Management in CCP

System-Level metrics: We implement both coarse- and fine-grained sampling solutions. We utilize Prometheus (https://prometheus.io/) to collect intra-MEC values in a coarse-grain fashion by exploiting both the built-in and custom node exporters. More specifically, we use the default node exporters coming with the official Prometheus' Helm chart to extract CPU, Disk, and memory utilization. We use the DCGM (https://developer.nvidia.com/dcgm) exporter for the near edge GPUs, which exposes a variety of metrics related to the card, e.g., temperature, memory usage, and PCIE throughput. However, currently, the delays in Prometheus' responses introduce noise in granular metrics extraction. That is, due to network delays (0-100ms) and scrape target's response handling, Prometheus would pose limitations in infrastructures hosting

latency-critical services. Thus, since tracing frequency is important for detecting spikes, we have designed and implemented custom fine-grained metrics extraction.

AIF-level metrics: The majority of the targeted services are latency-critical and require stringent adherence to user-specified Quality of Service (QoS) requirements. To continuously measure AIF performance, e.g., throughput and latency, we have implemented custom metrics scraping, local advertising, and aggregation of metrics. Similarly, to our system-level metric monitoring scheme, our AIF-level monitoring relies on a combination of coarse- and fine-grain data collection.

Additionally, to provide support for acquiring AIF-level metrics ad-hoc, we have included an endpoint on the AIFs. Each AIF saves an arbitrary number of its latest runs, and calling the metrics endpoint returns a list of the saved AIF-level metrics. In the following screenshot you can see an example return. The AIF returned to the client only the latest run's metrics, along with some information about the AIF.

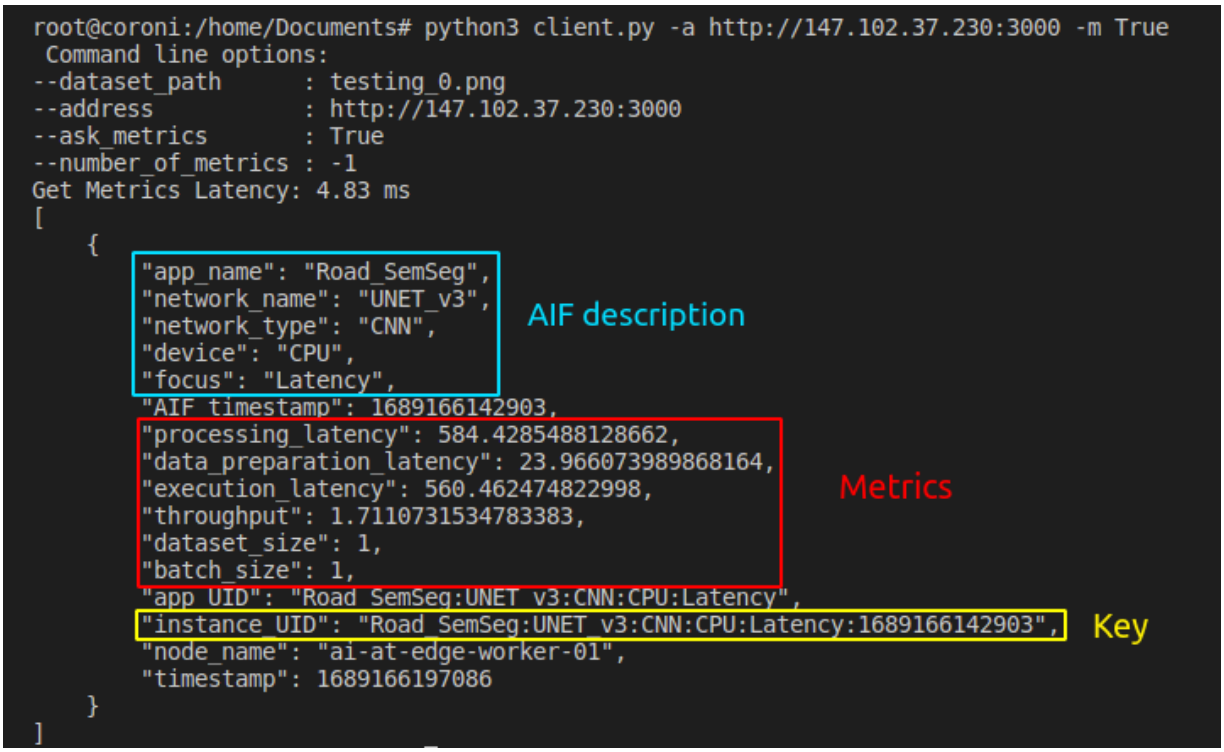


Figure 33 AIF-level metrics exposed to CCP, example with Image Segmentation

Metrics aggregation: Metrics that have been collected using fine-grained scraping are transferred to a local in-memory data store for intermediate storage. We utilized Redis-Timeseries key value store. This allows for low-latency write operations. Afterwards, our custom tool propagates the accumulated metrics and acts as a node exporter for the centralized Prometheus (MEC-level). Similarly, those metrics are propagated to the higher levels of abstraction using Thanos (https://thanos.io/). Metrics aggregation reduces both the volume of network traffic, as well as the storage allocated for monitoring purposes in AI@EDGE CCP, while providing the required telemetry per-tier. E.g., Resource managers of higher tiers make decisions

based on coarse grain metrics, while the local ones (MEC-level) leverage the fine-grained metrics for selecting the device to host any incoming AIF.

2.7.5 Security Aspects for HW accelerators at the far-edge

The security risks involved with utilizing hardware accelerators such as GPUs, FPGAs, TPUs, and VPUs have become increasingly common as these devices have become more widely adopted in cloud and embedded applications. These hardware accelerators are utilized for handling huge amounts of delicate data and executing intricate machine learning operations, making them susceptible to different types of threats and security breaches. With the growing popularity of hardware accelerators like GPUs, FPGAs, TPUs, and VPUs in various applications, especially in cloud and edge computing, the need to address the security issues related to their use has become increasingly important. These devices are used to process significant amounts of sensitive information and perform critical tasks, which makes them vulnerable to attacks and security breaches.

Unauthorized access to sensitive data

Attacks on hardware accelerators can be initiated through software vulnerabilities. While attackers cannot introduce new functions through techniques like buffer overflow or code injection, they can manipulate inputs such as configuration parameters or memory values to exploit design flaws. For example, attackers may take advantage of weaknesses in the accelerator controller to carry out different types of attacks [24]. Vulnerabilities in the System on a Chip (SoC) design can also be targeted by attackers.

An attacker could alter the system bus in a manner that allows them to execute unauthorized operations on the hardware accelerators. If proper security measures are not in place, this could result in compromised execution and potentially provide access to sensitive information, either through the output port or by exploiting the shared memory space. Hence, it is crucial to have adequate verification and protection mechanisms to secure the execution and output of hardware accelerators.

Even though the hardware accelerator's specifications may be secure, its actual implementation can still be vulnerable to physical attacks [25]. These attacks take advantage of the implementation's weaknesses and can compromise protection mechanisms. Side-channel attacks, such as power and timing attacks, are used to steal secret information from embedded devices and cloud servers. For instance, the Advanced Encryption Standard (AES) algorithm is mathematically secure but can still be vulnerable to physical attacks. To counteract side-channel attacks, hardware accelerators can enhance security, for example, by providing constant execution time to make timing attacks ineffective. Intel has already added AES-NI instructions as a solution [26]. Hardware accelerators need to be protected against various types of attacks, including fault-based and side-channel attacks. If not properly protected, a circuit separate from the rest of the processor can become an easy target for precise power side-channel attacks, leading to the compromise of key information.

Proposed testing and security functions

The focus here is on FPGAs and GPUs which are applied at near edge and far edge computing applications in which AI/ML operations are applied. One scenario could include the deployment of servers at Kernel-

based Virtual Machines (Ubuntu-Server) with the CPU/RAM resources specified above and deployed a Kubernetes cluster that consists of the aforementioned VMs and t, is able to manage the additional custom resources, e.g., GPUs (Jetson AGX) first, and FPGAs (Alveo U280+U200) at a second stage either for near or far edge applications. GPUs and FPGAs are employing AI/ML functions. Verification process and safety under attacks can be tested and potentially ensured if the virtual machines keep the guest operating systems patched and protect their environment just as we protect your physical machine. Consider disabling unnecessary functionality, minimize the use of the virtual machine console, and follow other best practices.

Especially in the case of FPGAs, their programming is the process of loading a bitstream file into the FPGA. A bitstream file contains the binary sequence for the FPGA design. Bitstreams are usually vulnerable to the same security threats as regular software including malware, unauthorized copy, and Intellectual Property (IP) theft. Cyber criminals can duplicate or reverse engineer FPGA applications by reading the internal memory, intercepting the bitstream, or removing the protective cover of the chip. On the bright side [27], application developers do not reveal their design to FPGA vendors or suppliers. As a result, attackers cannot discover any application-related information by attacking the FPGA vendor. In addition, FPGA manufacturers do not use metallization for programming. Thus, reverse engineering techniques where attackers identify the silicon mask layer by decapping the chip do not work.

In the case of GPUs [28], Jiang et al. [29] show a correlation timing attack on GPU to completely recover the AES key. They use 128-bit “electronic codebook” (ECB) mode AES encryption with T-tables, which uses 16B key to encrypt a 16B block. Each thread performs one block-encryption. They store the keys in the constant cache. All threads of a warp simultaneously read the same round-key. T-tables are stored in GlM since access to them leads to different memory requests which are serialized and hence, storing them in the constant cache would lead to resource wastage. They record time in two ways: clean measurement, where the warp execution time inside a kernel can be measured, and noisy measurement, where the latency of incoming/outgoing messages can be measured. ECB of different bit length are applied, and a security function (policy) is formulated. As in the case of Intel AES-NI, beyond improving performance, the new instructions help address recently discovered side channel attacks on AES. AES-NI instructions perform the decryption and encryption completely in hardware without the need for software lookup tables. Therefore, using AES-NI can lower the risk of side-channel attacks as well as greatly improve AES performance.

2.7.6 Setting up of the NetFPGA with the design of an ad-hoc South-Bound I/F.

In D4.1 we introduced how we plan to leverage NetFPGA SmartNICs for runtime network monitoring aimed at fueling the anomaly detection AIF framework (described in D3.1) against known attacks, as well as the Split-and-Merge CIDS [13].

We designed a custom protocol [14] allowing the control plane to remotely populate the board’s routing tables and query information on metrics computed by the board and stored in local registries, updated in switching runtime. We achieved this goal thanks to the implementation of a lookup table (LUT) external function that can be directly edited by the P4 pipeline running on the SmartNIC. This approach is a way to exploit programmable network hardware to accelerate the computation of metrics required by AI algorithms and do that at runtime scale.

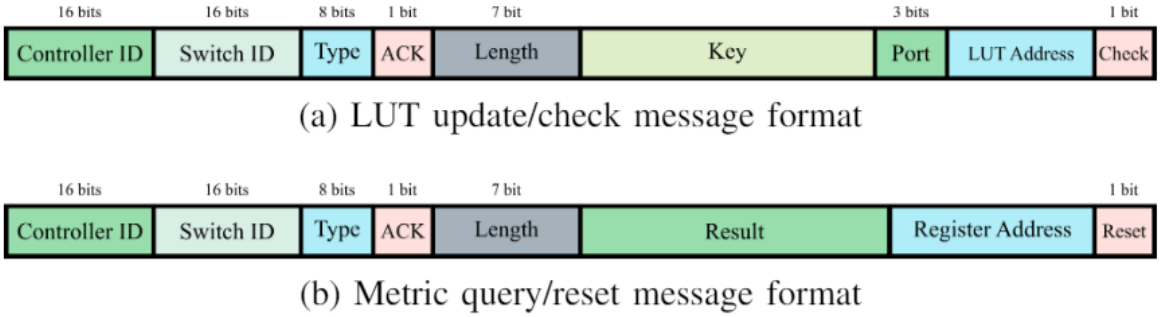


Figure 34 SBI control-plane message formats

Figure 34 shows the two control packet formats: LUT update/check (a) and metric query/reset (b), while Figure 35 displays a P4 program running on a standalone NetFPGA SUME Board. The data plane traffic triggers the metric update and is routed according to one or more lookup tables. A remote controller can use the southbound interface to interact with both stages of the pipeline. LUT update/check messages are used to populate the lookup tables and to inspect their status. Metric query/reset messages are used to read the content of the registers or to clear it.

The next step regards enhancing the scalability of the implemented LUT. In fact, even though instantiating lookup tables with hundreds or thousands of entries should be feasible in terms of memory footprint, it is not in terms of timing requirements. The size of the line encoder – i.e., a critical element of the LUT extern – scales linearly with the number of entries, causing the presence of negative slack on the critical path. We are currently working on a parallel implementation of the line encoder to mitigate this phenomenon.

The Service Based Interface SBI is currently using plain encapsulation over Ethernet frame, but this can be easily changed to upper layer protocols to favor interoperability, thanks to P4's flexibility in terms of protocol independence. This is one of the next steps of our activity.

After this, we will integrate Split-and-Merge metrics computation in the P4 pipeline, into a dedicated IDS SDN application. We will showcase the achievable performance in terms of attack detection latency and mitigation latency with two different use-case, the one of a large-scale 5G-MEC infrastructure (e.g., metropolitan/wide-area scale- and the one of a short-range MEC infrastructure (e.g., UC2).

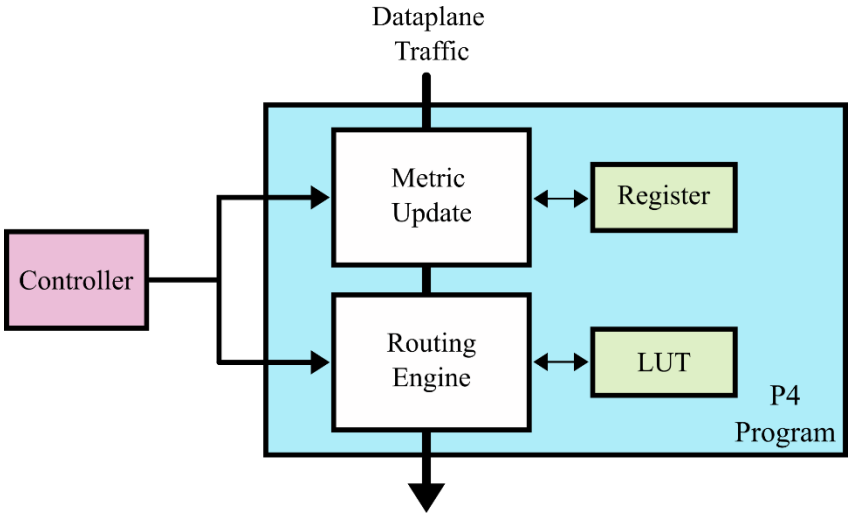


Figure 35 Data plane Overview

2.8 Orchestration Subsystem

The orchestration envisioned at the NSAP level is governed by the Multi-Tier Orchestrator (MTO), which can communicate with a multitude of orchestrators both at the cloud or at the network's edge. The MTO can issue orchestration operations (e.g., onboarding, deployment, termination, etc.) at an orchestrator located at the cloud or at any orchestrator located at the far or near edge, according to the requirements specified by the application. Conversely, the distributed MEC orchestration layout can manage the lifecycle and orchestration tasks of a local MEC system (together with all its elements, including but not limited, to the MECPM, the Serverless platform, the hardware acceleration modules, etc.). Once an AIF or MEC application is deployed, the MEC system is meant to fully carry out its functional operations.

2.8.1. Main features of the components

Figure 36 depicts the main orchestration and system components. The main features and design choices of the MEC orchestrator (MEO) can be identified as follows:

- Capability to communicate in a bi-directional manner with a higher-end Multi-Tier Orchestrator (MTO), located at the NSAP, and responsible for managing several types of orchestrators at both the edge(s) and the cloud. This communication takes place over the defined MTO-to-MEO interface through a messaging system (broker). It supports two types of communication to cover a set of required functionalities and use cases:
    - Broadcast: used to discover new available MEC systems and for service assurance in the MEC systems when needed.
    - Direct (RPC) communication: used as an imperative command to deploy, migrate or delete applications on a particular MEC system. This decision is computed by the Intelligent Orchestrator Component (IOC), which belongs to the decision-making module of the

network automation closed-loop stated in D3.1. The IOC decision is based on the availability of certain resources and services and transmitted to the MTO.

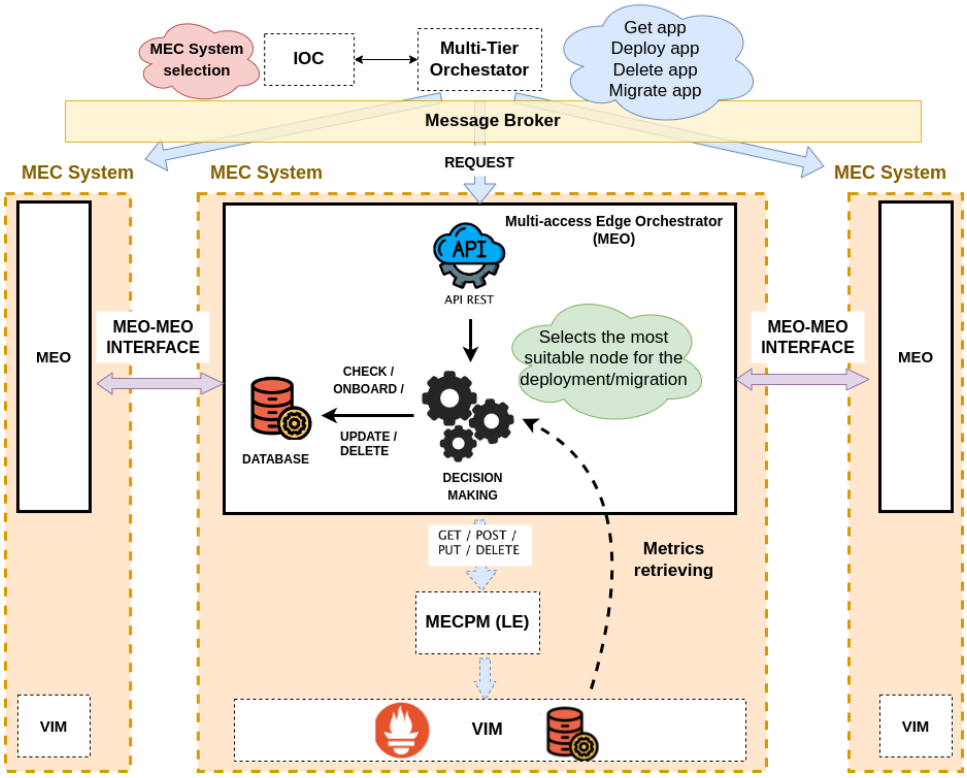


Figure 36Main Orchestration and system components

- Onboarding of AIFs and MEC applications, storing the descriptor files received from the MTO in a file system. Consequently, the MEO has a registry of any AIF or MEC application running in that MEC system.
- Capability to communicate in a bi-directional way with the IOC within the MEC system, which can access the aggregated data of the local MEC system.
- Capability of deploying AIFs and MEC applications on a specific near/far edge. In the scenario that the deployment needs to be deployed in a specific place for reasons of location or both hardware or computational resources, the orchestrator can receive those requirements and performing the deployment in the required location (edge).
- Distributed and bi-directional communication with other MEOs if the MTO at the NSAP is not functional or available. This communication occurs through the defined MEO-to-MEO interface through a messaging system (broker), which closely corresponds to the ETSI GS MEC 040 standard [16]. This standard encompasses a federated broker component designed to facilitate seamless connections between orchestrators situated in distinct domains, providing added value in terms of

interoperability and inter-domain orchestration capabilities. The MEO-to-MEO interface facilitates operations like discovering new MEC Systems and migrating applications across different MEC Systems through a messaging system (broker). While the current interface offers a subset of functionalities compared to the standard, future developments could implement additional APIs to achieve a fully distributed architecture. These additional APIs would enhance communication and coordination between MEOs, promoting a more comprehensive and decentralized system. The current implementation provides two types of communication to cover the following functionalities and use cases:

  - Broadcast: a MEO uses a messaging queue to determine if certain MEC systems have the resources to migrate applications, for instance, in case of being close to an SLA breach. To do so, the MEO would request its neighbours the current status of the resources demanded by the requirements of the applications to be instantiated, e.g., in terms of RAM memory, hardware acceleration requirements, etc.
  - Direct (RPC) communication: it is required when the IOC at MEO level selects a specific MEC system to migrate an application because its requirements are no longer met. This method is only used in the case that the migration procedure through the NSAP cannot be carried out.

- Lightweight, scalable, secure and portable characteristics because the logic of the module has been encapsulated in a docker image that can be deployed as a container. This gives the MEO the characteristics and benefits of containerization.
- Created to be Kubernetes (K8s) based and Helm-compatible because as it was stated in D3.1 [2]: "*in the AI@EDGE project, K8s performs the role of Virtualized Infrastructure Manager (VIM) and Helm charts could be used by the platform as descriptors for managing K8s packages and their operations*". Furthermore, the MECPM (LightEdge) and several Serverless platforms such as Nuclio, only accept the Helm-chart format.
- Connection with a software application (Prometheus), which is used for event monitoring and alerting. It records real-time metrics in a time series database that can be accessed by the Orchestrator and the IOC in order to make decisions for the placement and migration scenarios.

2.8.2. StandAlone and Non-StandAlone MEC Orchestrator

In order to ensure that the orchestration systems can be in more heterogenous scenarios, in which we could find networked systems composed only of edge layers, and of both edge and cloud tiers, the design of the MEO has been performed in such a manner that it can work both in StandAlone (SA) and in Non-StandAlone (NSA) mode with respect to the NSAP. The NSA mode will be utilized when the MEC orchestrator is part of the full architecture and will act as a second-tier orchestrator with full connection to the NSAP. For the scenario where a MEC System is isolated and has no connectivity with the NSAP, the SA MEO will act as the entrance of the system and be the main managing and controlling the network. This feature will allow the architecture to be highly distributed and decentralized. Moreover, this feature makes it possible for the platform to adapt to various system deployments, depending on the technological

characteristics and security requirements (e.g., to avoid any connection from outside premises) of the specific use cases in WP5. The modularity introduced by the design of the NSAP, MTO and MEO enables the composition of different versions of the platform to accommodate to the necessities of the particular scenarios used for deployment.

Figure 37 depicts a software composition proposed for the MEO SA deployment. This version of the MEO is meant to be used isolated from the NSAP, therefore allowing all the processes of the MEC system to function in the same way they would do if the NSAP was present. Different from the MEO NSA, the MEO itself becomes here the entry point of the orchestration system. In other words, the API exposed by the MEO is the one employed to perform any operation of the lifecycle management, including onboarding, deployment, etc., for that specific MEC system together with the MECPM. Notice that in this case, this version of the MEO is meant to manage the activities of a single MEC system, due to any restrictions in the deployment. For that reason, no communication with any other MEO is envisioned (i.e., the MEO-to-MEO interface is not present). This version of the system counts with its own telemetry manager (Prometheus is used in the concrete example of the figure), which is managed internally within the MEC system (which in this case is deployed on a single K8s system) by the MEO.

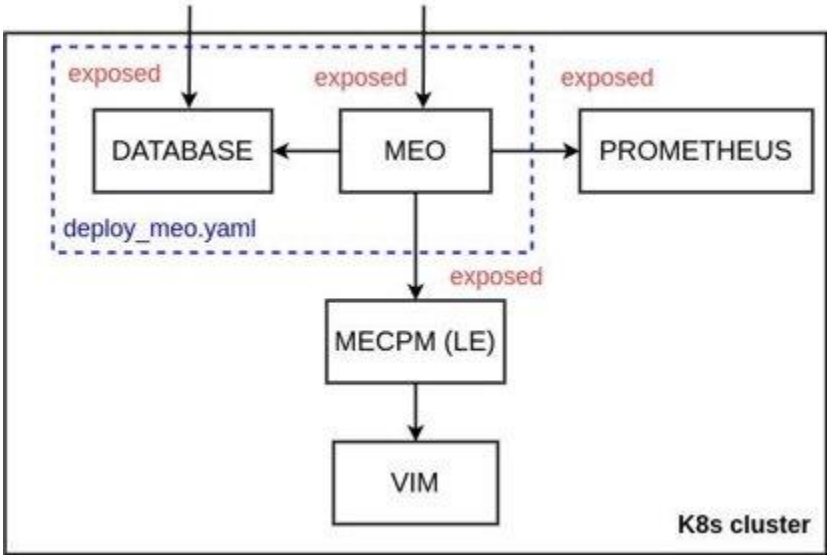


Figure 37 Example of software composition for MEO SA

The previous scenario envisioned a case in which due to several constraints a connection with the main NSAP at the cloud was not possible, and the deployment was essentially composed of a single MEC system. Nonetheless, this privacy, security, or any other extra requirements to avoid a remote connection, could also be present in a scenario where multiple MEC systems are meant to be managed and working together. For this reason, in AI@EDGE we have introduced a simplified version of the NSAP (named mini NSAP), which is meant to be deployed in any local infrastructure with internal connection to the various orchestrated MEOs. The mini NSAP provides the basic orchestration capabilities of the full NSAP, including the MTO, the IOC, the AIFD database, as well as the message broker that enables the MEO-to-MEO and the MTO-to-MEO communications. An exemplary software composition of the mini NSAP (in the MEO NSA

version) is depicted in Figure 38. Different from the previous structure, it is possible to observe that the IOC is in this case fed from the data pipeline enabled from the telemetry data coming from all MEC systems. In this particular case, a Thanos agent is used to concentrate the data from the telemetry exporters deployed from each MEO NSA. As is the case of the main NSAP, the API interface of the MTO is exposed as the main entry point of the platform.

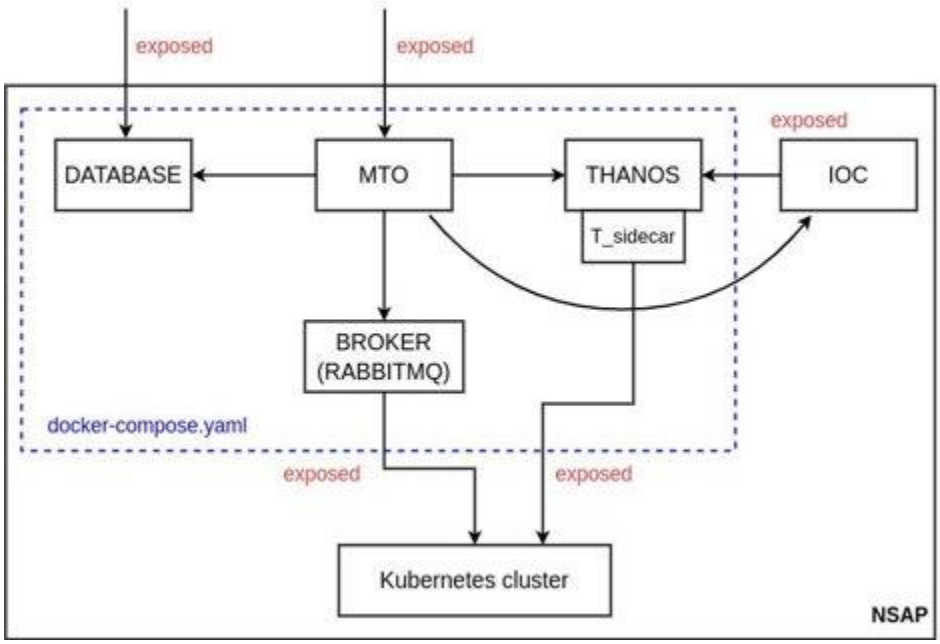


Figure 38 Example of software composition of the mini NSAP for MEO NSA

As previously mentioned, the NSAP data pipeline referred to infrastructure telemetry is exported by each MEC system through a local metric exporter. In this specific case, the implementation is based on Prometheus. However, this role could be played by other software implementing the same functionality, due to the modularity of the components. Once this data is exported, the sidecar deployed for each MEC system together with the main gathering process (e.g., Thanos), can jointly collect and properly manage it for the whole platform in a scalable manner. This organization can be observed in Figure 39.

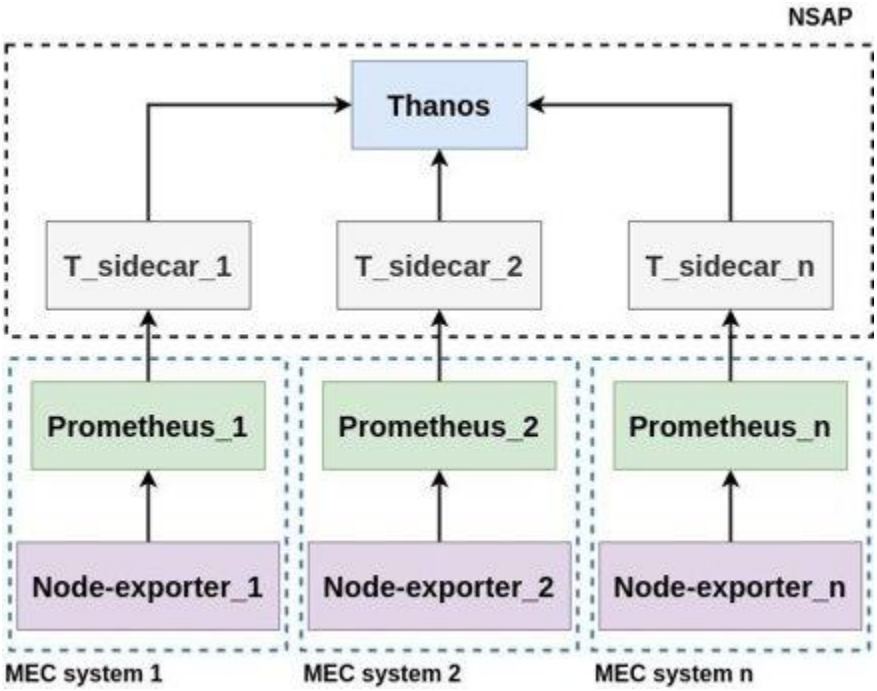


Figure 39 Internal structure of the infrastructure telemetry data for the NSAP and mini NSAP

2.9 Integration of the CCP

2.9.1 Architecture

This Section details the concrete implementation of the CCP, which has been studied from the point of view of the two main ETSI architectures mentioned in D4.1: ETSI MEC in NFV and ETSI MEC. The consortium has thoroughly analyzed the advantages and disadvantages of adopting each architecture, especially paying attention to the requirements and objectives of the project. The main insights can be summarized as follows:

- *ETSI MEC in NFV*. This reference architecture has been adopted in a multitude of European projects and has facilitated bringing closer the cloud and the telco worlds. Consequently, there exist very well-known orchestration and management tools, such as Open-Source MANO (OSM), that could be leveraged to build and deploy the modules of AI@EDGE. While this would facilitate the deployment due to the availability of tools and the alignment and collaboration with other projects, it provides less flexibility to introduce new concepts and modules that were not meant at the time the ETSI MEC in NFV architecture was introduced. On the one hand, the applications and services are deployed as VNFs, using a certain descriptor, depending on the selected tool. This would make the customization of the AIF descriptor difficult, the addition of new fields and the employment of AIF graphs, as first detailed in D4.1. This would determine not only the specification of the descriptors but also the limited room for adaptation on the orchestrators available, which would be rather limited for extension on the lifecycle management necessary to attend the AIFs' needs. On

the other hand, AI@EDGE introduces in the architecture new modules aligned with the edge-to-cloud continuum, such as the Serverless platform, and taking the road towards future 6G networks, such as the inclusion of the data pipeline and the lifecycle management of the AIFs. For that reason, the current design and support of the ETSI MEC in NFV architecture limits the room for new research on these topics and its native adoption in the network layout.

- *ETSI MEC.* As opposed to the previous one, this reference architecture aims to provide a more general vision of the infrastructure and a lighter orchestration approach, enabling greater flexibility to integrate the different functionalities envisioned in the project, such as the Serverless platform, and meet the requirements of AI@EDGE. However, it is true that the availability of open-source software implementing this orchestration stack is rather limited, and the majority of the software available is commercial, has restrictions in the adoption of RAN and core protocol stacks, or makes difficult to introduce new capabilities required by the NSAP and the CCP, which difficult the reusability of existing software from other 5G-PPP projects.

In view of the above, and in order to be able to cover the aforementioned functionalities, the consortium has advocated for selecting a more general architecture, as ETSI MEC is, regardless of the availability of existing software, and ensure that the capabilities introduced are generic and reusable for other new projects that could leverage the research performed in AI@EDGE. For that reason, following the requirements stated in D2.1 and D2.2, the AI@EDGE project has proposed the design of a two-level orchestration approach: at the NSAP (centralized) and at MEC system level (distributed). Figure 40 shown below depicts the updated version of the integration in the CCP taking as a reference the ETSI MEC architecture, and updating the vision presented in D4.1. With respect to D4.1, it is possible to highlight the following differences:

- **far edge.** Firstly, being considered low-computation nodes, the far edges do not harbor any more the functionalities corresponding to the MEC Platform Manager (MECPM). Note that the MECPM is implemented using the LightEdge platform, as stated in D4.1. By contrast, these entities are just placed on the associated near edge and communicate to the MEC Platform hosted in the far edge a via standard ETSI interface (mm5). Consequently, the VIM of each far edge is governed also by the MECPM at the near edge (via standard mm6 interface). Moreover, even if on a smaller size and capacity on what can be found in the near edge, the MEC hosts of the far edges also may incorporate hardware acceleration capabilities, which are used by the applications according to the deployment options.

- **near edge.** On the one hand, the MEC host at the near edge presents the same view of the one at the far edge, with the only exception that the hardware acceleration capabilities count with greater computational power. On the other hand, the Intelligent Acceleration Resource Management (IARM), which cooperates with the MEO, providing the additional functionality required to leverage the hardware accelerators' capabilities; and the Serverless Platform have been incorporated in the vision presented in this deliverable D4.2. As stated in Section 0 the Serverless Platform has been deployed at the near edge, introducing a new interface towards the MECPM, which allows it to issue the deployment of new Serverless functions, following the same workflow of any other application. This interface is implemented via a REST API (implementing CRUD operations), enabling the possibility to deploy, gather, and delete Serverless functions with respect

to MECPM. Conversely, the IARM implements the intelligence to be provided on orchestration activities. For that reason, it must interact with the MEO in order to provide, based on the status of the network, timely decisions on the applications management. These decisions include deployment decisions when a new request is received, in case the AIF's requirements include hardware acceleration, migration decisions within the same MEC system if the requirements of a running application are not met anymore, etc. The interface defined between the IARM and the MEO is based on a REST API, allowing the MEO to request the IARM, for instance, the optimal hardware placement decision through a POST procedure. Similarly, the IARM could also use REST POST procedures to request the migration of an AIF from one node to another. Finally, the hardware acceleration modules are interconnected with the VIM in order to provide the capabilities required when deploying one or more AIFs with those requirements over the virtualized infrastructure. The accelerated containers will ultimately be deployed by Kubernetes and gain access to their respective HW acceleration resources (GPU, FPGA), as described in Section 3.3.3, which will allow them to interface with REST APIs as ordinary endpoints exposing their service to users and interact with VIM as ordinary AIFs.

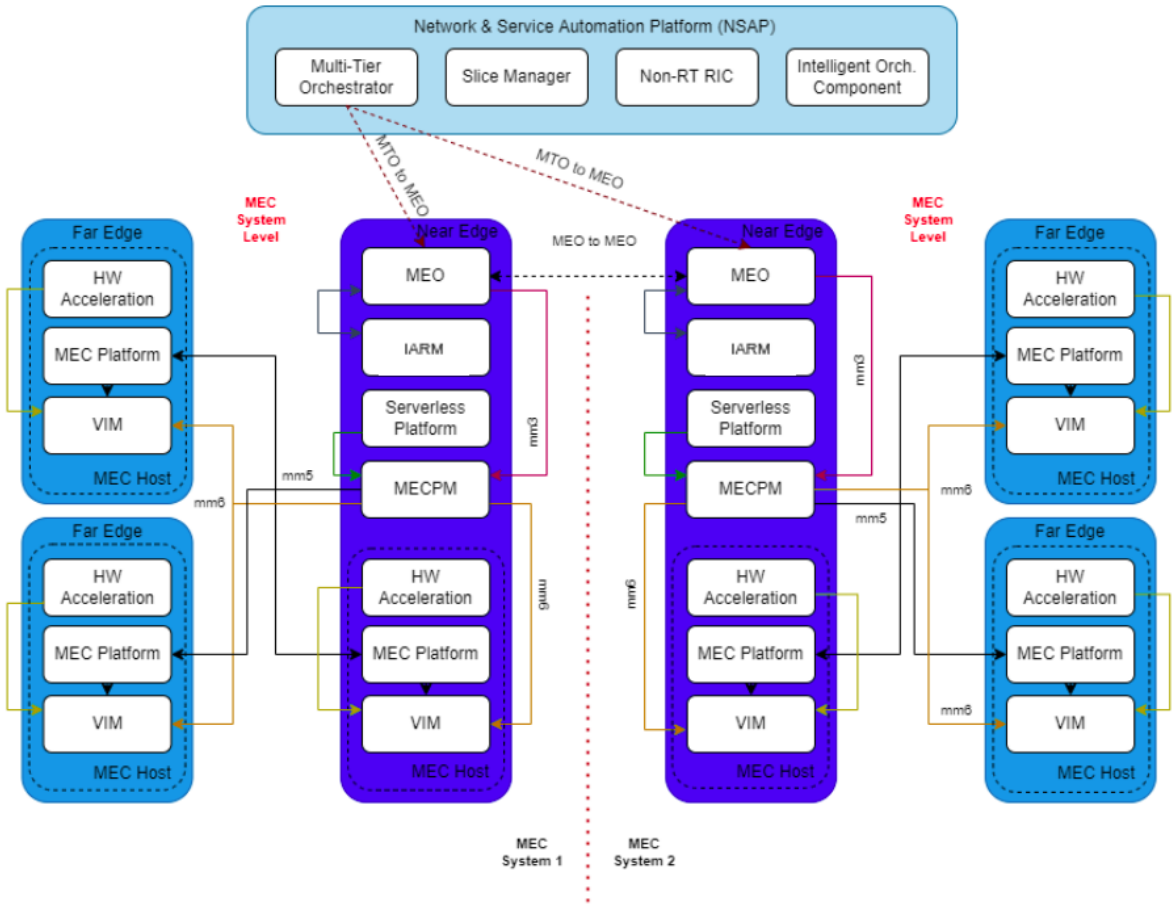


Figure 40 Specific implementation of the AI@EDGE architecture based on the ETSI MEC reference architecture

2.9.2 MEC Workflows

This Section depicts the main workflows defined at the moment of writing this deliverable D4.2. Section 2.9.2.1 introduces AIF deployment with hardware accelerators workflow and Section 2.9.2.2 introduces AIF migration with hardware accelerators workflow.

AIF Deployment with hardware accelerators

The process envisioned within the AI@EDGE architecture for application deployment is detailed in Figure 41. Three main groups of modules are depicted, namely NSAP, MEC System 1 and MEC System 2. The NSAP site comprises the orchestration capabilities (Multi-Tier Orchestrator) in this layer (comprising an NFV Orchestrator and VIMs considers that the PDU session of the UE requesting the application instantiation has been previously established and is out of the scope of this process); and the decision-making module (IOC).

The instantiation process is triggered from the Operations Support System (OSS) when receiving a request to instantiate an application. As a result, the request reaches the entry point of the NSAP, whose role is

played by the Multi-Tier Orchestrator (MTO). Then, the MTO module retrieves the AIF requirements specified in the descriptor file and provides them to the IOC. Having this information together with the status information of each underlying system (e.g., available hardware, resources, and services), this module takes an intelligent decision on the placement of the application. This decision will indicate to which node of a specific MEC system the application must be forwarded through the MTO-to-MEO interface or if it should be allocated into the cloud the first case, the IOC selects the near edge node of the MEC System 1 to instantiate the AIF/app. The decision is returned to the MTO, which forwards it to the MEO of the selected MEC System. Before the application is instantiated, the application package must be onboarded to the system. This process is triggered because of the application instantiation request from the OSS and is enabled by the MEO at the MEC System or, by contrast, the NFVO at the cloud. After that, the MEO sends a request to the IARM module, which provides additional functionality required to leverage the hardware accelerators' capabilities on the selected node. If it has more than one kind of hardware accelerator, the IARM will select the most suitable for the AIF deployment. Furthermore, the MEO receives the response from the IARM, and the request is forwarded to the target MEC Platform Manager to initiate the process. Besides the application requirements, the descriptor received by the MEC Platform Manager contains the traffic rules to be set on the target MEC platform. After this, the application is ready to be deployed, and to this end, the MECPM interacts with the VIM to proceed with the deployment.

In the second case, the process follows a similar workflow as the one described above but on this occasion the IOC selects the MEC System 2 to instantiate the AIF/app.

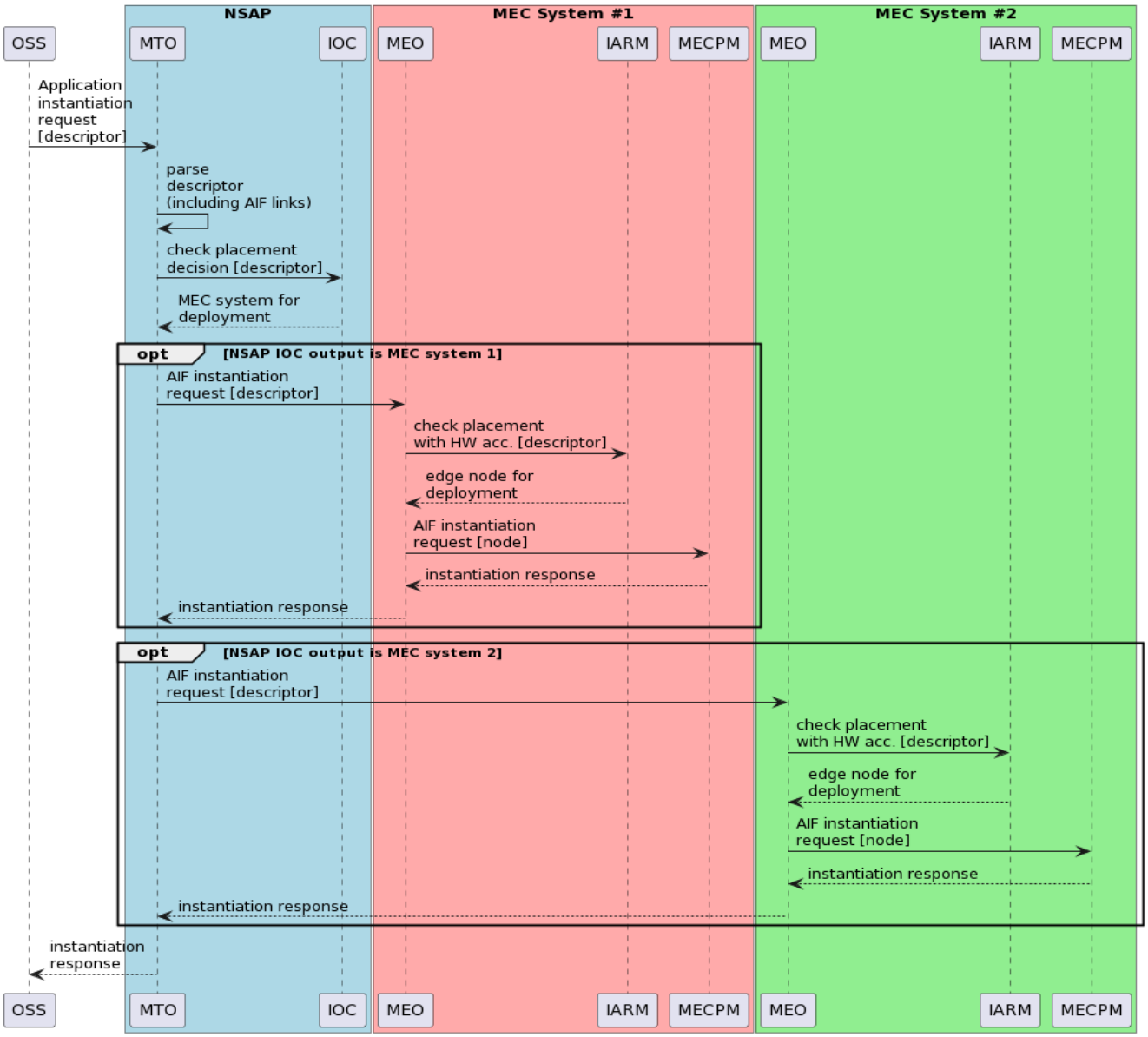


Figure 41 Workflow showing the application deployment process

AIF migration with hardware accelerator

This section details the workflow considered for application migration processes. In particular, two main cases are distinguished when the hardware where application is running presents low efficiency: (i) scenarios where another hardware accelerator is available within the same MEC system and (ii) scenarios where no suitable candidate is found within the MEC system, and therefore the application is migrated to another MEC system. Figure 42 depicts both scenarios in parallel.

On the top of the workflow, the application is migrated within the MEC system, where the MEC host located at both near and far edges share the same UPF. Therefore, in this scenario, a UPF reselection is not involved. This workflow is triggered when the IARM raises an alarm indicating that the hardware where the application is running presents low efficiency. In the figure showcasing this scenario, it is assumed that initially the application is deployed at Node 1. In this case, the MEO receives a migration request from the IARM to migrate the AIF to the Node 2, which has more suitable hardware accelerators. The MEO forwards the request to the MECPM, and the same procedure followed for application deployment in the previous section is followed. Consequently, the application will be onboarded in the VIM, and the procedure for instantiation will be carried out, including the DNS and traffic rules configuration to ensure that the UE traffic is properly routed. After receiving the confirmation from the target MEC Platform Manager, the MEO can request the original MEC Platform Manager to terminate the application, remove the traffic rules and free the allocated resources at the VIM.

On the bottom part of the workflow, the second migration case envisioned is showcased, in which the MEC system where the application is currently deployed has no suitable hardware accelerators to satisfy its requirements. In the specific example depicted in the figure, the application is deployed at the MEC System 1 and is migrated to the MEC system 2. To do so, the MEO would inform the MTO of the need for migration through the MTO-to-MEO interface. Then, the MTO will forward the request to the IOC module, which will determine the most suitable MEC System depending on the hardware requirements. Once the decision is reached the MTO forwards the decision to the destination MEO of the MEC system selected. It communicates with its local IARM, which will determine the most suitable node depending on the hardware accelerators available and the specific requirements of the application. After this procedure, the MEO has the required information to request the MECPM the instantiation of the application. Once the AIF has been deployed and running the source MEO will proceed with the termination process and free the reserved resources at the VIM.

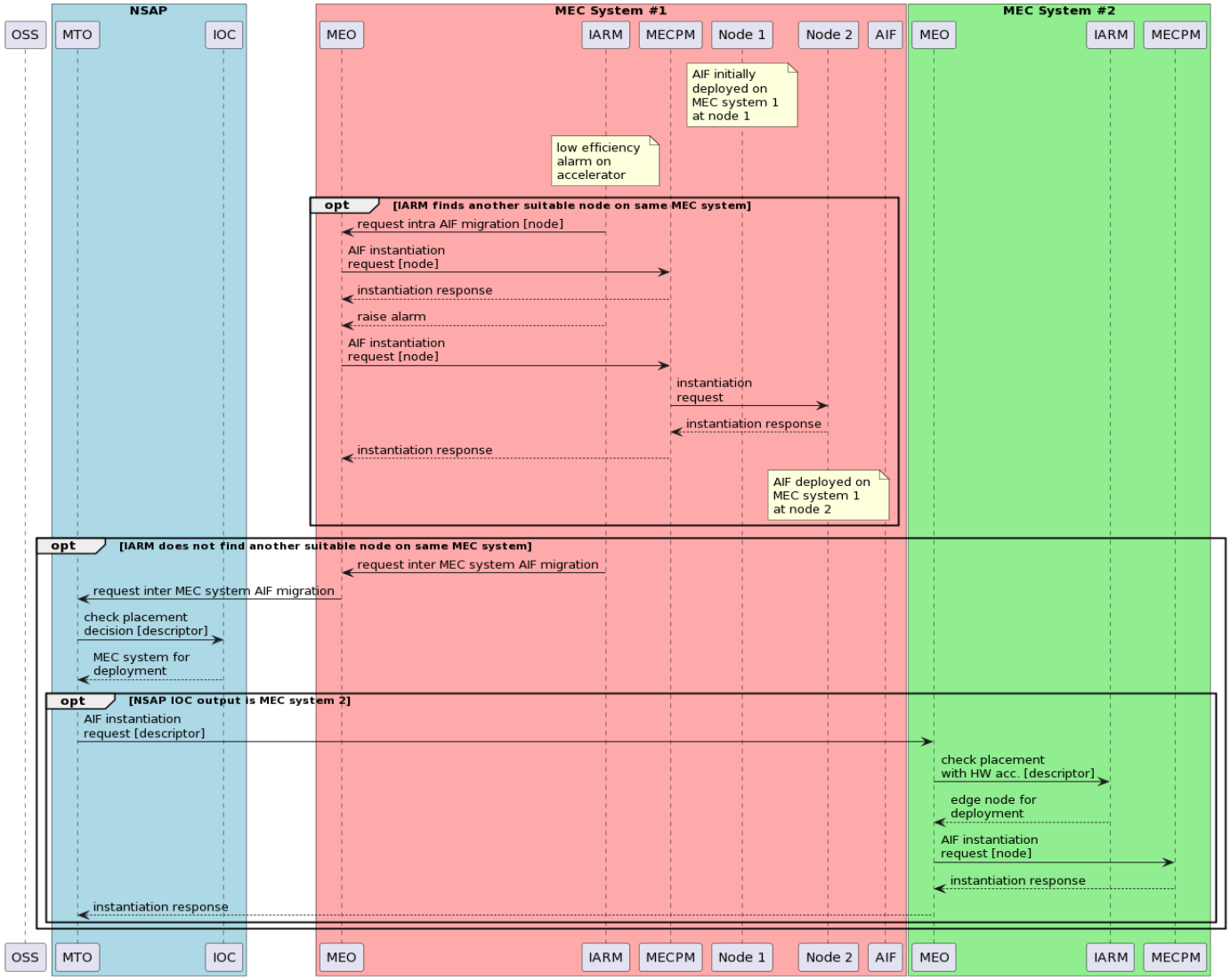


Figure 42 Workflow showing the application migration process

3 Validation methodology for the Connect-Compute Platform

The current section describes the methodology that has been used for validating the CCP, i.e., the testbeds developed, and the "scenarios" devised to evaluate each CCP component/feature, as well as the entire integrated CCP. The following subsections include details regarding the setup of two testbeds, i.e., the reference (FBK premises) and the auxiliary one for acceleration studies (ICCS site), as well as steps/plans and metrics for testing each CCP feature.

3.1 Reference Testbed

The Integration testbed hosts the main components of the system. As shown in Figure 43, the testbed is based on a Kubernetes cluster that spans both near and far edge nodes.

- near edge host is connected to an N6 network interface of the Athonet 5G SA Core, it hosts the master node of the cluster, together with the LightEdge Platform and Nuclio Platform and hosts Applications and Serverless Functions. Co-located with the near edge host, the MEO platform is also deployed here.
- far edge host is connected to an N6 network interface of the Athonet 5G SA Core, and hosts Applications and Serverless Functions.

3.1.1 Prototype architecture

Near edge and far edge are connected through a management network, that allows the synchronization between Kubernetes components.

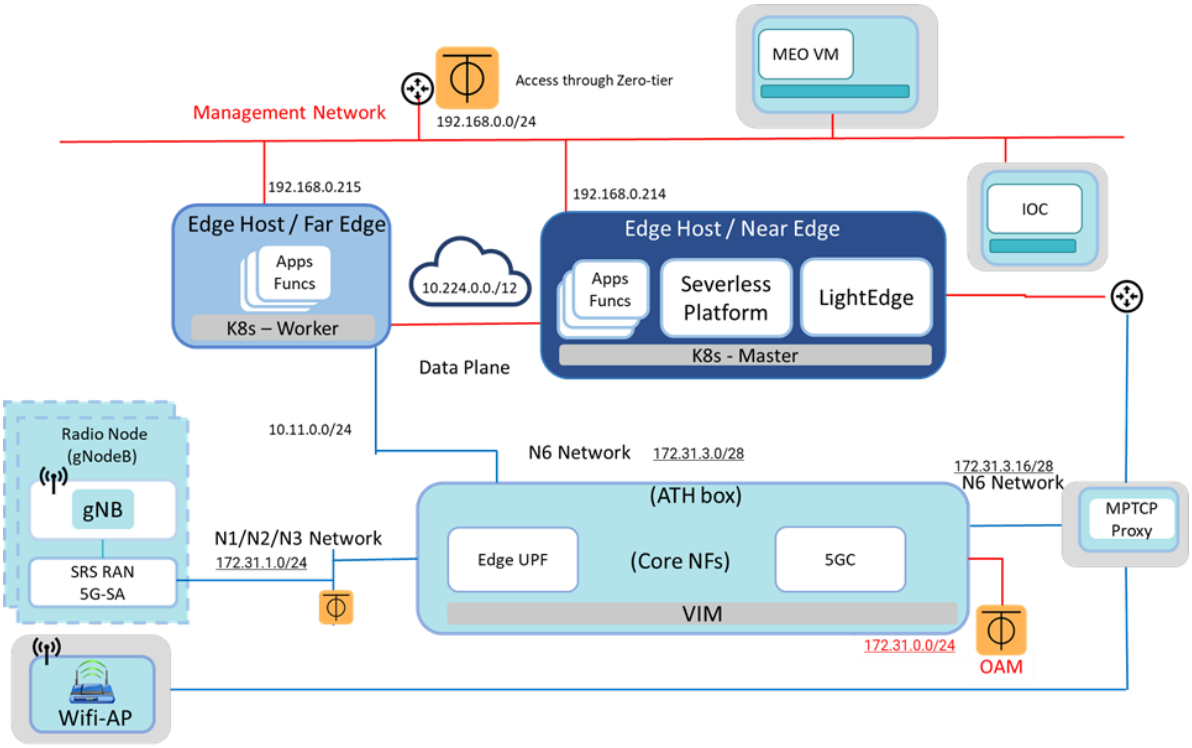


Figure 43 Integration Testbed Main Components

The architecture also includes the components related to the Multipath TCP experimentation: the Wi-Fi AP, the MTPCP proxy.

3.1.2 Deployment details

The testbed is instrumented with metrics provided by Kubernetes and 5GC. Both the Kubernetes components and Athonet core emit metrics in Prometheus format. This format is structured plain text, designed so that people and machines can both read it. Prometheus Server or some other metrics scraper can be configured to periodically gather these metrics and make them available in some kind of time series database. In most cases metrics are available on /metrics endpoint of the HTTP server. Metrics services are activated in the k8s cluster in Testbed. A Graphana server is also available for metric visualization.

Remote connection towards the testbed is allowed through a virtual network enabled by the ZeroTier service [15] This tool combines the capabilities of VPN and SD-WAN and emulates Layer 2 Ethernet with multipath, multicast, and bridging capabilities. The testbed uses two ZeroTier L2 networks, enabling two different access points: one through the management network, that allows the interaction with Kubernetes nodes and orchestration components; one connected directly with the 5GC.

Also, the Athonet's 5GC collects several Key Performance Indicators (KPI) available, that are exported in Prometheus format.

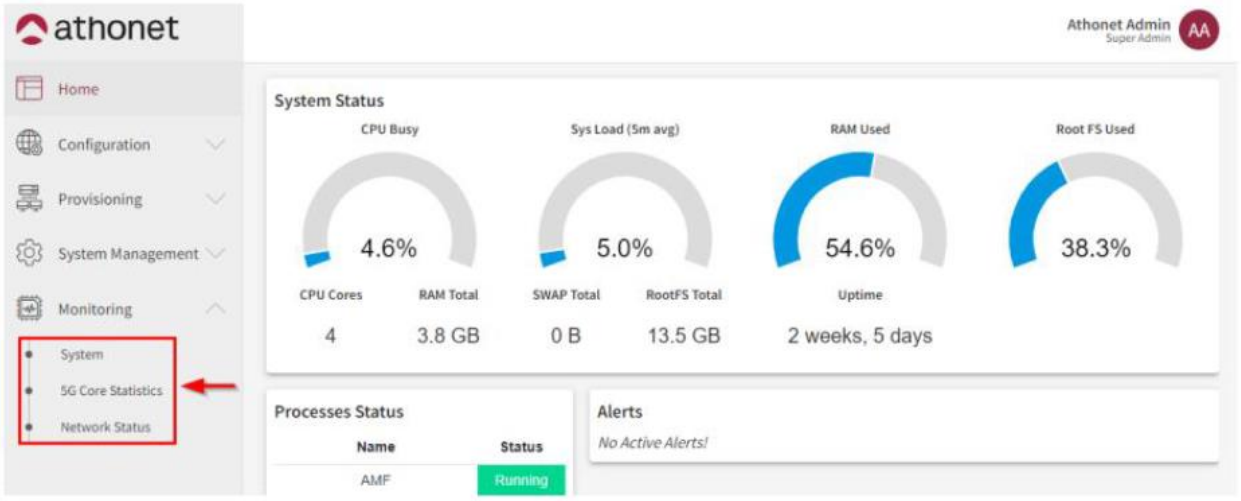


Figure 44 5GCore GUI showing metrics

The 5GC system contains a built-in Prometheus instance which offers several ways to retrieve and visualize its collected metrics. These KPIs are grouped into two sections:

- System KPIs, which report metrics related to the performance of the node on which the 5GC is running (e.g., System Health, Internal processes). The system KPIs provide information on: Boot process, Filesystem information, Timing, CPU performances, Incoming/outcoming traffic, System errors, Memory performances, Node load, Network interfaces, Disk performances, System caching, Network statistics, Context switch, Memory swap, Network errors.
- 5GC statistics KPIs, which are the metrics related to the specific 5GC functionalities. Core metrics are associated to the different 5G network functions: Access and mobility Management Function (AMF), AUthentication Server Function (AUSF), Network Repository Function (NRF), Policy Control Function (PCF), Session Management Function (SMF), Unified Data Management (UDM), Unified Data Repository (UDR), User Plane Function (UPF) (Table 2).

Table 2 5GC statistics KPIs

Network Function	**KPI**
SMF	Number of PDU sessions
	SMF NAS 5GS messages received/sent
	SMF NAS 5GS messages decoding failure s
	Number of SUPI in the SMF
	Number of Session in the SMF
	Number of DNN supported by the SMF
	Number of configured/allocated IP addresses
UPF	Number of sessions
	Number of GTP-U interfaces
	Number of IP interfaces
	Number of forwarded packets/bytes

	Number of dropped packets/bytes
AMF	Number of internal crash in handling UEs/RANs
	NGAP RRC establishment causes
	NGAP messages received/sent
	NAS 5GS messages received/sent
	NAS 5GS messages decoding failure
	Number of UEs in the AMF
	Number of active UE connections
	RAN node status
UDR	Number of provisioned SUPIs
	Number of provisioned data profiles
PFCP	PCFP messages received/sent
	PFCP messages decoding failure
	PFCP node number of detected reset
	PFCP node number of retransmission detected/performed
	PFCP node number of timed-out requests
	PFCP node status
	PFCP node local/remote recovery timestamp
	PFCP peer allocated tunnels
PCF	Number of N7 sessions
	Number of N5 sessions

3.2 Acceleration Testbed

Besides the reference testbed of AI@EDGE at FBK premises, we have set up a secondary testbed at ICCS premises specifically for the study of accelerators in MEC scenarios. This acceleration testbed also serves as an intermediate step towards the integration of accelerators to the complete CCP. This secondary cluster is connected to the reference testbed via VPN link.

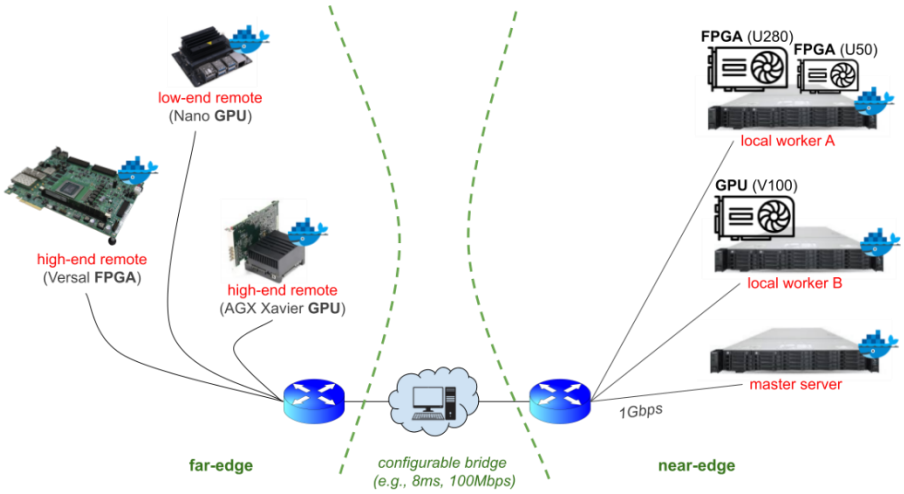


Figure 45 Auxiliary testbed for MEC acceleration studies

ICCS has built a local cluster at its own premises in order to study the accelerators' performance and develop custom solutions for AI@EDGE. The key idea is to utilize a number of servers and accelerators to emulate edge computing scenarios involving multiple nodes of diverse compute capabilities each, to test various integration approaches, to study orchestration techniques, and measure AIF deployment efficiency, all while developing certain FPGA/GPU code to accelerate representative AIFs of AI@EDGE. Figure 45 shows a high-level description of the cluster. We use three supermicro Intel Xeon servers and five self-contained embedded processors. The latter five can emulate "remote" nodes at the far edge, whereas the former three can emulate nodes at the near edge with increased computational power (or even a single near-edge cluster/cloudlet). The Ethernet interconnections of these nodes may vary in terms of bandwidth to support various network scenarios. The three supermicro servers will represent three different cases: a server with FPGA acceleration, a server with GPU acceleration, a server without acceleration. Furthermore, we will also use one of the servers to assume multi-card acceleration scenarios on a single node by equipping a server with 2-3 FPGA cards. We have deployed Virtual Machines on top of the aforementioned servers (PMs) to emulate a multi-tenant, virtualized environment. Regarding the "remote" nodes, we have utilized an ultra-low-power GPU-based SoC and the latest state-of-the-art FPGA-based SoC for diversity purposes.

In our case, near and far edge are meant to be two computing tiers, with varying network proximity, e.g., reduced network-related latencies for the far-edge devices, and diverse computing characteristics. To better emulate the computing continuum on our local cluster, we have segmented our local network into two parts. The first subnetwork in which the near-edge, more powerful, high-end servers are connected, remained the same. On the other side, we have connected the far-edge nodes to a 100Mbps switch which in turn is connected to an intermediate node before connecting to our local network. This node is a PC, in which we installed a second NIC card, created a bridge between two physical network interfaces and added additional network delay. Therefore, any UE equipment will connect to the far-edge nodes with decreased latency

(through the switch), while connecting from the far-edge/UE to the near-edge costs more in terms of latency, since an additional hop, over the link with the increased network latency is required.

More specifically, the key characteristics of the 7 nodes in the ICCS testbed-cluster are:

- Intel Xeon E5-2658A without accelerators (8 CPUs, 8GB RAM). Also, to be considered as the master of the local cluster (Ubuntu 20.04.1 LTS).
- Intel Xeon Gold 6138 with 1 GPU (8 CPUs, 16GB RAM, Nvidia GPU Tesla V100). To be considered as a worker in the cluster (Ubuntu 20.04.1 LTS).
- Intel Xeon Silver 4210 with 2 FPGAs (4 CPUs, 16GB RAM, Xilinx Alveo U200, Xilinx Alveo U280). To be considered as a second worker in the cluster (Ubuntu 18.04.6 LTS).
- Xilinx Versal AI Core Series VCK190 Evaluation Kit. To be considered as a high-end remote node utilizing the latest FPGA acceleration technology (ACAP with AI engines)
- NVIDIA Jetson Nano Developer Kit. To be considered as a low-power low-end remote node with GPU acceleration (Ubuntu 18.04.6 LTS).
- NVIDIA Jetson AGX Developer Kit. To be considered as a low-power low-end remote node with GPU acceleration (Ubuntu 18.04.3 LTS).
- Zynq MPSoC UltraScale+ ZCU 104 Kit (x2). To be considered as a low-power remote node with FPGA acceleration.

The aforementioned HW is supported by certain OS+SW components. As described in Section 2 (Hardware Acceleration Solutions for AI/ML), on top of the Intel Xeon servers we have deployed Kernel-based Virtual Machines (Ubuntu-Server) with the CPU/RAM resources specified above and deployed a Kubernetes cluster that consists of the aforementioned VMs and the rest devices of the ICCS testbed. Kubernetes, with the appropriate plugins installed, can manage the additional custom resources, e.g., GPUs, and FPGAs.

We have further updated the Kubernetes cluster, adding the low-powered devices with integrated accelerators, residing in the far-edge. While neither the vanilla Kubernetes, nor the hardware vendors, i.e., NVIDIA (Jetson Xavier), Xilinx (MPSoC Ultrascale+ ZCU 104) provided software for seamless integration, we have created and advertised extended, custom resources for those nodes. Therefore, Kubernetes API can treat those resources similar to the already existing ones, being aware of whether they are occupied or available.

3.3 Testing Methodology

To evaluate each feature/component of the CCP, we methodically lay down a respective “scenario”, i.e., the testbed involved, the final objective, the metrics, and the steps to be taken, all in the form of a script/plan. The following subsections present a scenario per CCP component, as well as a master scenario for the integrated CCP. The results of these scenarios will be presented and analysed in the next section with experimental data. The following table summarizes the information about the different scenarios that aim at covering the whole set of CCP functionality.

Table 3 CCP functionality evaluation scenarios

ID	Scenario	Description	CCP Components	NSAP Components
3.3.1	MTO, MEO, Interfaces & AIF deployment and migration	Validate the onboarding and deployment of AIFs from the NSAP to the specific distributed edges, including the functionalities expected from the MEO once it is integrated as part of the distributed CCP. Furthermore, to validate the robustness and the new Standalone (SA) and non-Standalone (NSA) features of the MEO in different scenarios.	MEO NSA, LightEdge, K8S	MTO, Thanos, Message broker
		Validate the capacity of CCP to onboard and deploy simultaneous requests. As a result, this scenario demonstrates the capability and robustness of the SA MEO to deploy AIF when multiple simultaneous instantiation requests are made.	MEO SA, LightEdge, K8S	-
		Evaluate the stateless distributed migration procedure through the MEO-to-MEO interface with simultaneous migration requests (in case of MTO is unavailable or standalone architecture is considered) across five different MEC systems.	MEO NSA, LightEdge, K8S	Message broker
3.3.2	MPTCP: benefits	Evaluate the benefits of multi-path aggregation at the transport layer in multi-connectivity multi-RAT scenarios	MPTCP Proxy/Server, gNB, UEs	-
	MPTCP: Predictive scheduling	Validate and evaluate the benefit of the predictive scheduler: The predictor analyzes metrics and tries to anticipate packet loss, and delay variations before they are notified via TCP signaling. It then updates the MPTCP proxy/server and in turn, the MPTCP scheduler decides to change the downlink interface, duplicate the packet or adapt scheduler interface weights.	MPTCP Proxy/Server, gNB, UEs	-
3.3.3	Acceleration1: benefits of	Evaluate the benefits of using HW acceleration in the MEC, evaluate the resource management efficiency (assignment of incoming AIFs to suitable HW nodes)	IARM, accelerators	
	Acceleration2: integration and seamless use in CCP	verify IARM and MEO and CCP component integration, seamless use of HW acceleration in the CCP	MEO, MECPM, IARM, accelerators	MTO, IOC
3.3.4	Serverless: AIF serving	Evaluate the use of serverless technologies for AIF serving. Based on the re-usable model servers and standard APIs. Evaluate qualitatively (i.e., functionality prototype) and quantitively (deployment/update time)	LightEdge, K8S, Serverless (Seldon Core)	-
3.3.5	Integrated CCP	IOC validation and evaluation for AIF placement on different MEC systems	MEO, LightEdge, K8s	IOC, MTO

3.3.6	NSAP: Non-RT RIC RAN slicing SLA assurance	rAPP to manage slice resource allocation in multi-RAT scenarios (cellular and Wi-Fi) to comply with SLAs in time and space. The objective is to demonstrate non-RT RIC closed-loop optimisations in multi-rat environments and its main functionalities (e.g., rApps, ICS, data exposure, SMO operations)	Disaggregated RAN	non-RT RIC, SMO
3.3.7	LCM: Model management and model update	Prototyping and evaluation of the AIF LCM related to model re-training, publishing, and AIF update	MEO, LightEdge, K8S, Serverless	MTO, Model registry, Thanos
3.3.8	LCM: auto-configuration	A meta-learning based solution for the auto-tuning of AIF	\-	-
3.3.9	LCM: Model monitoring	Continuously monitor the performance of the AIF models to detect cases of drift or outliers in the input/output data	MEO, LightEdge, K8S	MTO

3.3.1 Scenario1: MTO, MEO, Interfaces & AIF deployment

- Objective: validate the onboarding and deployment of AIFs from the NSAP to the specific distributed edges, including the functionalities expected from the MEO once it is integrated as part of the distributed CCP. Furthermore, to validate the robustness and the new Standalone (SA) and non-Standalone (NSA) features of the MEO in different scenarios.
- Testbed: despite acknowledging the existence of two distinct testbeds, FBK and ICCS, it is pertinent to elucidate that i2CAT's was designated for "in-lab" prototyping and development of solutions, before integration with further components. i2CAT's testbed comprises the NSAP located in a cloud server, and five MEC systems composed of three edge nodes (one near edge node, and two far edge nodes).
- Scenario 1.1: the scenario aims to validate the capacity of the NSAP and CCP to onboard and deploy simultaneous instantiation requests across five different MEC systems. Consequently, this scenario demonstrates the capabilities and robustness of the MTO and the NSA MEO while also validating the MTO-to-MEO interface. This scenario assumes that some entities such as the MTO, message broker, MECPM (LightEdge) and VIM (K8s cluster) with the metric retriever server are already up and running.
- Scenario 1.2: the scenario aims to validate the capacity of CCP to onboard and deploy simultaneous requests. As a result, this scenario demonstrates the capability and robustness of the SA MEO to deploy AIF when multiple simultaneous instantiation requests are made. This scenario assumes that some entities such as the MECPM (LightEdge) and VIM (K8s cluster) with the metric retriever server are already up and running.
- Scenario 1.3: this scenario aims to evaluate the stateless distributed migration procedure through the MEO-to-MEO interface with simultaneous migration requests (assuming that the MTO is down) across

five different MEC systems. This scenario assumes that some entities such as the MECPM (LightEdge) and VIM (K8s cluster) with the metric retriever server are already up and running.

- Metrics:
    - Onboarding and deployment time of simultaneous AIFs through the MTO and message broker (Scenario 1.1).
    - Error rate of AIF's requirements fulfillment (Scenario 1.1, 1.2 & 1.3).
    - Onboarding and deployment time of simultaneous AIF(s) through the SA MEO (Scenario 1.2).
    - Full migration for when the MTO is not working (s) (Scenario 1.3).
    - MEO required resources for the deployment and migration experiments (CPU & RAM) (Scenario 1.2 & 1.3).
- Steps scenario 1.1:
    - Enable a wall time measurement tool at the MTO when an onboarding or deployment operation is received. This timer must be activated when the operation is issued at the northbound of the MTO. If the AIF is not previously onboarded, the MTO will save the timestamp once this operation is completed and before the deployment procedure is continued. After the AIF deployment is finished, the corresponding MEO informs the MTO about it and the measurement will be stopped, accounting for the time taken by all entities involved including MTO, MEO, MECPM and VIM.
    - At NSAP level deploy the MTO module together with the message broker (used for the MTO-to-MEO interface) and the metric aggregator server.
    - When the MTO is deployed, the AI@EDGE AIFD repository is retrieved and onboarded automatically in a file system at the MTO container. Each AIF descriptor file has a yaml format and contains, among other parameters, the minimal computational resources required to deploy the AIF. The AIFD files are provided by the developers.
    - At every MEC System the MEO module is deployed together with the metric retriever server. Also, launch the metric exporter in every node of the system.
    - When the MEOs are deployed, the AI@EDGE repository is onboarded automatically, which contains the AIFs logic in helm format, uploaded by the specific developers.
    - Verify that the metric aggregators at the MEC systems and at the NSAP are properly enabled and configure.
    - Prepare the deployment REST call to the MTO for each of the tested AIFs, including the MTO endpoint and the name of the AIF descriptor file to be deployed.
    - The call previously described will effectively trigger: (i) the onboarding of the AIFD in the MTO file system (in case it is not already there); (ii) the check of the MTO to the NSAP IOC to verify the node on which the AIFD shall be deployed; (iii) the communication between the

MTO and the corresponding MEO to request the deployment of the AIF in a specific node handled within its system; and (iv) the communication with the MECPM and the VIM for the resource allocation and instantiation of the AIF. However, these steps are transparent for the user and are triggered automatically when the MTO receives the request.

- The MTO will receive a response from the corresponding MEO based on the output of the AIFD deployment. The counter of erroneous/successful requirements is then increased accordingly, and the timer to account for the deployment time is stopped.

- Steps scenario 1.2:
  - Enable a wall time measurement tool at the MEO when an onboarding or deployment operation is received. If the AIF is not previously onboarded, the MEO will save the timestamp once this operation is completed and before the deployment procedure is continued. After the AIF deployment is finished, the timer will be stopped, accounting for the time taken by the entities involved including MEO, MECPM and VIM.
  - When the MEO is deployed, along with metrics exporter in every node of the system, the AI@EDGE AIFD repository is retrieved and onboarded automatically in a file system at the MTO container. Each AIF descriptor file has a yaml format and contains, among other parameters, the minimal computational resources required to deploy the AIF. The AIFD files are provided by the developers. The AI@EDGE repository is onboarded automatically, which contains the AIFs logic in helm format.
  - Prepare the deployment REST call to the MEO for each of the tested AIFs, including the MEO endpoint and the name of the AIF descriptor file to be deployed.
  - The call previously described will effectively trigger: (i) the onboarding of the AIFD in the MEO file system (in case it is not already there); (ii) the internal check of the MEO to verify the node on which the AIFD shall be deployed; and (iii) the communication with the MECPM and the VIM for the resource allocation and instantiation of the AIF. However, these steps are transparent for the user and are triggered automatically when the MEO receives the request.
  - The MEO will receive a response based on the output of the AIFD deployment. The counter of erroneous/successful requirements is then increased accordingly, and the timer to account for the deployment time is stopped.

- Steps scenario 1.3:
  - Enable any kind of wall time measurement tool at the MEO when a migration operation is received. This timer must be activated when the operation is issued from the Local IOC to the origin MEO. After the new AIF deployment is finished, the destination MEO informs the origin MEO about it and it starts the deletion of the original AIF. The time measurement will be stopped, accounting for the time taken by all entities involved including MTO, IOC, MEOs, Local IOC, MECPM and VIM.

- At every MEC System the NSA MEO module is deployed providing the necessary information regarding the IP and ports of the MECPM, message broker, and metric retriever server.
- When the NSA MEOs are deployed, the AI@EDGE repository is onboarded automatically, which contains the AIFs logic in helm format, uploaded by the specific developers.
- When the Local IOC detects that the node allocating the AIF no longer meets the resources requirements, it will effectively trigger the migration request to the MTO through the MTO-to-MEO interface. If the connection is successful, the request is forwarded to the NSAP IOC, which will select the optimal destination node for the AIF migration. On the other hand, if the connection is not achieved, the MEO will send a broadcast request for the minimal computational resources required to every MEO registered in the message broker. The Local IOC will retrieve the information from the MEOs and select the optimal destination node for the AIF migration. The communication between the origin MEO and the destination MEO to request the migration is done through the MEO-to-MEO interface.
- The communication with the MECPM and the VIM for the resource migration and release of resources are transparent for the user and are triggered automatically when the Local IOC launches the request.
- The origin MEO will receive a response from the corresponding MEO based on the output of the AIFD deployment. The counter of erroneous/successful requirements is then increased accordingly, and the timer to account for the deployment time is stopped.

3.3.2 Scenario2: MPTCP

- Objective: To test multipath aggregation at the transport layer in multi-RAT multi-connection scenarios and evaluate the capabilities of MPTCP predictive.
- Features evaluated: Multi-path traffic, Throughput, Link handover/Failure, MPTCP predictive scheduler
- Steps: Run each test in turn while recording the metrics from UE and server sides
  - Test 1: Test with two links equivalent in terms of performance:
    - The client connects to WiFi and cellular network at the same time. Two links equivalent in terms of performance
    - The client connects to the server and begins downloading a large file
    - Measure metrics during the test:
    - Throughput over time
    - Number of packets lost/duplicate
    - Evaluate metrics: Compare the client's throughput to the maximum throughput of each connection

- Test 2**:** Test with two links with different delay or bandwidth:
  - The client connects to WiFi and cellular network at the same time. Two links have different delay or bandwidth
  - The client connects to the server and begins downloading a large file
  - Measure metrics during the test:
  - Throughput over time
  - Number of packets lost/duplicate.
  - Evaluate metrics: Compare the client's throughput to the maximum throughput of each connection
- Test 3**:** Test in case one of the links is degraded:
  - The client connects to WiFi and cellular network at the same time. Two links equivalent in terms of performance;
  - The client connects to the server and begins downloading a large file;
  - The WiFi connection degrades due to reduced throughput and increased latency
  - Measure metrics during the test
  - Throughput over time
  - Number of packets lost/duplicate
  - Evaluate metrics: Analyze the client's throughput before and after the connection degrades.
- Test 4**:** Test in case one of the links is failed:
  - The client connects to WiFi and cellular network at the same time. Two links equivalent in terms of performance;
  - The client connects to the server and begins downloading a large file
  - One WiFi connection has failed
  - Measure metrics during the test
  - Throughput over time
  - Number of packets lost/duplicate
  - Evaluate metrics: Analyze the client's throughput before and after the connection degrades.
- Test 5: Test in case one of the links is restored after failed/degraded:
  - The client connects to WiFi and cellular network at the same time. Two links equivalent in terms of performance;
  - The client connects to the server and begins downloading a large file

- One connection degrades due to reduced throughput and increased latency
- After a few minutes, the link is restored.
- Measure metrics during the test
- Throughput over time
- Number of packets lost/duplicate
- Evaluate metrics: Analyse the client's throughput before, during and after the connection degrades.

- Test 6: MPTCP predictive scheduler
  - The client connects to WiFi and cellular network at the same time. Two links equivalent in terms of performance;
  - The client connects to the server and begins downloading a large
  - Wifi network become unstable. Continue test in two case:
  - With xAPP: xAPP predict and change the MPTCP scheduler behave.
  - Without xAPP
  - Measure metrics during the test
  - Throughput over time
  - Number of packets lost/duplicate
  - Evaluate metrics: compare the case with xAPP and case without.

3.3.3 Scenario3: Acceleration

- Objective: validate the seamless use of HW acceleration in the CCP and evaluate its benefits
- Scenario: assume multiple incoming AIF requests to the CCP, with 1-4 AIF acceleration versions each (distinct image containers for far-, near-, FPGA-, and GPU-based executables)
- Testbed: the acceleration testbed (ICCS premises)
- Metrics: Equivalent functionality & accuracy to original AIF (yes/no), Execution Latency (msec), Execution Throughput (outputs/sec), HW Initialization Overhead (msec),, Power consumption (Watt), Energy (Joule)
- Steps:
  1. Select multiple representative AIFs (CNN, LSTM, MLP, ML)
  2. Implement various accelerated versions per AIF

- Based on the hardware: CPU-only, GPU V100, Alveo U280, Alveo U200, Zynq MPSoC, Xavier AGX, Xilinx Versal, Jetson Nano
- Based on the optimization goal: latency, throughput, power

3. Integrate the binary developed for bare-metal (not virtualized) environments, with a wrapper to allow for service invocation over the network
4. Build the final image and push it on the AI@EDGE registry
5. Create/Emulate different AIF descriptors based on
   - Application Name
   - Network Type, e.g., CNN, LSTM, MLP, ML
   - Optimizations Goals: latency, throughput, energy
   - A list of the available Helm charts that include an image that implements the AIF (found in the AI@EDGE provided catalogue).
6. Create scripts for assumed incoming AIF requests and available HW accelerators
   - e.g., two successive requests for CNN AIFs with FPGA-only acceleration
   - e.g., an AIF request for far-edge when all far-edge devices are occupied
   - e.g., an AIF with both FPGA- and GPU-based acceleration but distinct goals
7. Execute a script, call IARM from Dummy MEO/IOC to make decisions (all automated)
   - e.g., examine available helm charts and current CCP/MEC/testbed state, leverage an internal Look-Up-Table to select proper accelerator to meet user requirements
8. Measure metrics during script execution (with carefully placed timers & monitors/logs)
   - Measure IARM latency
   - Measure HW accelerator initialization
   - Measure execution Latency, Throughput, Power, Energy, per AIF
   - Validate AIF result correctness
   - Assess conflicts in resource allocation, deadlocks, fallback choices
9. Evaluate metrics
   - accelerator vs nominal execution (FPGA/GPU vs CPU)
   - accelerator vs accelerator (e.g., GPU vs FPGA, near- vs far-edge computing)
   - compare IARM decision and ensuing AIF performance to optimal scenario
   - validate correct operation and benefits for CCP

10. Return to step 6 (continue validation with new scripts, until all cases have been tested)

Notice that steps 1-5 are part of the development phase (described to a certain extent in Section 2.7).

3.3.4 Scenario4: Serverless Platform

- Objective: The goal is to measure the overall deployment time for a functional and responsive AIF service. This will provide insights into the performance of various deployment scenarios and highlight the advantages of serverless architecture over manual image building methods. The serverless approach offers several advantages over manual image building methods, including automated model deployment, automatic scaling, and reduced operational overhead. By using Seldon Core as a serverless framework, we can easily deploy and manage an AIF service without worrying about the underlying infrastructure and model serving, making it an ideal choice for automated and low maintenance AIFs deployments.
- Scenario: In the general scenario considered, an ML/DL model is periodically updated. The deployed model is the same structurally (e.g., a MultiLayer Perceptron with the same number of layers and the same hyperparameters), but parameter values change. We will use this scenario to test the deployment time of different AIF services. The concrete deployment cases analysed are two:
    - The first case involves deploying an AIF service that includes the ML model directly into the container. The model is an MLFlow model exposed as a service using MLServer. The image building is not automated.
    - The second case involves deploying a Seldon Core deployment that downloads the model at bootstrap and exposes the model as a service using a pre-built image provided by Seldon Core.
- Testbed: FBK testbed with one MEC system composed of two edge nodes.
- Metrics:
    - Time of image building.
    - Time of service deployment.
    - Resource usage in terms of image space and computational resource usage.
- Steps:
    - Save a generic machine learning model in a MLFlow format on a MinIO storage system.
    - Create two different deployment plans using Helm charts: one for the manually built image and one for the Seldon deployment.
    - Measure the time used to build the image.
    - Deploy the AIF services through MEO.

- Measure how long it takes for the deployment to be ready by using a Python script to check the status of the deployment. This helps us understand how long it takes for the AI function to be available for use.

3.3.5 Scenario5: Integrated CCP

- Objective: validate the correctness of AIF deployment in various MEC setups, validate the integration of different CCP and NSAP components, mainly focusing on the deployment of AIFs with or without hardware acceleration requirements. We focus on MECPM, MEO, IARM, IOC, and MTO, all working together. To gradually reach our goal, we utilize 3 distinct MEC configurations and we divide the validation into 3 distinct tests.
- Scenario: assume multiple incoming AIF requests to the CCP, with 1-4 AIF acceleration versions each (distinct image containers for far-, near-, FPGA-, and GPU-based executables) and deploy automatically
- Metrics: AIF Deployment Overhead (msec), AIF Deployment Correctness, AIF E2E Latency (msec), AIF Throughput (fps)
- Tests/steps:
  1. Test1: test individual MEC systems, testbed = 2 separate MEC (FBK, acceleration at ICCS)
     - Step1: Setup the FBK cluster (1 master node, 1 CPU worker node).
     - Step2: Deploy CPU version AIF in the FBK CPU worker node, performing either a segmentation or a classification task
     - Step3: Take deployment overhead and correctness measurements in 3 different configurations in the MEC System: not using MEO or IARM, using only MEO, using both MEO and IARM
  2. Test2: test single large MEC system including very remote nodes, testbed= FBK + ICCS accelerator
     - Step1: Append the remote accelerator node from ICCS to the FBK Cluster, so that the final cluster consist of 1 master node, 1 CPU worker node and 1 “remote” FPGA worker node
     - Step2: Deploy in the ICCS remote accelerator node: a CPU only version AIF performing either a segmentation or a classification task, an FPGA version AIF performing a segmentation task
     - Step3: Deploy a single AIF performing a segmentation task: an FPGA AIF in the remote ICCS accelerator node, a CPU AIF in the remote ICCS accelerator node, a CPU AIF in the FBK CPU worker node.
     - Step4: Repeat same deployments for an AIF performing a classification task.
     - Step5: Take the E2E Latency and Throughput measurements of the AIFs.
  3. Test3: test entire CCP with two MEC systems under same NSAP layer, testbed= Cloud, FBK, ICCS

- Step1: Deploy monitoring mechanism in both MEC systems so that NSAP can get aggregated values from both, in order to decide initial MEC placement of incoming AIF. IOC leverages high-level MEC metrics in order to decide which MEC System to deploy the AIFs to.
- Step2: Configure and deploy IOC to make placement of AIFs in the wanted MEC System (ICCS or FBK)
- Step3: Deploy segmentation/classification CPU/GPU/FPGA AIFs in the selected MEC System.
- Step4: Take the Deployment overhead and correctness measurements.
- Step5: Repeat Steps 2-4 changing the MEC system selected.

3.3.6 Scenario6: non-RT RIC

- Objective: The main objective is to evaluate the capabilities of the non-RT RIC to expose RAN monitoring and control to rApps.
- Scenario6.1: The scenario comprehends the non-RT RIC and a variable number of rApps producing and consuming data stored in a Prometheus server.
- Testbed: non-RT RIC plus the rApps and the data lake.
- Metrics: Excess delay (measured interval vs expected interval) while obtaining the data by the consumer rApps. (ms)
- Steps:
    - Deploy a producer rApp gathering data from a Prometheus server.
    - Deploy a consumer rApp obtaining data through the producer rApp. Measure excess delay.
    - Increase the number of consumers and repeat excess delay measurements.
    - Increase the number of producers and repeat steps 2 and 3.
- Scenario6.2: The scenario comprehends different RAN slices deployed in a multi-RAT environment (Wi-Fi, 4G and 5G). An rApp is deployed to produce the RAN telemetry data exposed by the RAN nodes and stored in a Prometheus Server. Another rApp is designed to manage the allocation of resources (i.e., airtime or PRBs) in each RAN node according to the actual utilization and to the resources that were required during slice deployment. This rApp will consume the exposed data, compute a resource allocation, and notify it to the RAN nodes through the non-RT RIC.
- Testbed: non-RT RIC plus the rApps, data lake, the RAN nodes (Wi-Fi, 4G, 5G) and the UEs.
- Metrics:
    - RAN slicing rApp functionality (yes/no)

- RAN slicing rApp error in resource allocation (required per slices vs obtained) (%). Comparison with static allocation.

- Steps:
  - Deploy RAN slices in the available RATs with the initial static allocation of resources.
  - Connect UEs and start different traffic patterns to evaluate static resource allocation.
  - Start RAN slicing rAPP. Connect UEs and repeat traffic patterns to evaluate dynamic resource allocation.

3.3.7 Scenario7: LCM: Model Management and Model Update

- Objective: The goal of this validation scenario is to determine the time taken to deploy a working and responsive AIF service. In traditional serverless deployment solutions, updating the model requires redeploying the entire function, which can be time-consuming and can cause downtime for the function. However, with the sidecar auto-updater architecture, the process of updating the model is automated and seamless. This means that the AIF function can be updated without any downtime, and the new model can be made available to users immediately. This validation scenario shows that the serverless sidecar architecture provides better timing results in terms of time savings and continuous service delivery compared to the bare serverless architecture.
- Scenario: In the general scenario considered, an ML/DL model is periodically updated. The deployed model is the same structurally (e.g., a Multi-Layer Perceptron with the same number of layers and the same hyperparameters), but parameter values change. We will use this scenario to test the deployment time of different AIF services. The concrete deployment cases analysed are two:
  - The first case involves deploying a Seldon Core deployment on the VIM. The deployment downloads the model at bootstrap and exposes it as a service.
  - The second case involves deploying a Seldon Core with sidecar architecture on the VIM. The sidecar performs periodic checks for the availability of new models and executes automated updates of the model as a service by substituting the outdated model with the new one without service downtime.

- Testbed: FBK testbed with one MEC system composed of two edge nodes.
- Metrics:
  - Time of service deployment;
  - Time of service redeployment after model update.
- Steps:
  - Save a generic machine learning model in a MLFlow format on a MinIO storage system.
  - Create two different deployment plans using Helm charts: one for a simple Seldon deployment and one for a Seldon deployment with a sidecar architecture.

- Deploy the AIF services through MEO.
- Measure how long it takes for the deployment to be ready by using a Python script to check the status of the deployment. This helps us understand how long it takes for the AI function to be available for use.
- Update the model.
- Measure how long it takes to update the machine learning model service for each deployment plan. For the simple Seldon deployment, it is just a matter of redeploying the updated model. For the Seldon deployment with the sidecar architecture, the update happens automatically without the need for redeployment.

3.3.8 Scenario8: LCM: Auto-configuration

- Objective: The objective of this validation scenario is to assess the time required for configuring an AIF (Artificial Intelligence Framework) and to evaluate its performance based on the specific configuration. In the conventional approach, configuring an AIF necessitates repetitive re-training until the optimal configuration is achieved, resulting in time-consuming and disruptive downtime for the function. On the other hand, the auto-configuration solution automates and streamlines the model update process. As a result, the AIF function can be configured without any downtime, and the new model becomes instantly available to users.
- Scenario: Proof-of-concept method [32]. We first consider our AIF as an intrusion detection function that classifies each network flow based on whether it is benign or an attack. Our goal isn't to create a new intrusion detection AIF but to properly configure an existing one. Hence, we've chosen a Random Forest classifier as our AIF. We focus on two hyperparameters to configure: (1) the number of features to consider each time to make the split decision, and (2) the minimum number of samples required to be at a leaf node. For the evaluation data, we use two public datasets divided into 17 days, with each day containing both benign traffic flow and attack flow. Attack types differ daily, with flow distribution skewed towards benign traffic. In this scenario, every day is treated as an individual dataset.
  This Proof-Of-Concept will also undergo evaluation utilizing the dataset obtained from the testbed associated with use case 2 of the WP5 work package. To compare our method based on meta-learning, we have two configuration baselines: (1) Scikit-learn's default settings and (2) optimal settings obtained by using Bayesian Optimization (BO). Using BO, we aimed to maximize the MCC as the target metric. Regarding meta-learning, to configure one day, we use all the other days to construct our meta-dataset and then our meta-model to infer our configuration. This strategy is applied in a rotating manner across all days, yielding 17 results corresponding to the 17 days that make up our datasets.
- Testbed: Experiments were carried out on a single core of an Intel Xeon CPU E5-2650 0 @ 2.00GHz.
- Metrics:
  - The time required for configuration.
  - The performance of the AIF is evaluated using precision, recall, and MCC metrics.
  - Resource usage in terms of memory space and computational resource usage

- Steps:
    - **Meta-dataset Creation**: Using all days except the target day to create a meta-dataset.
    - **Meta-model Construction**: With the data from the meta-dataset, build a meta-model.
    - **Configuration Inference**: Use the meta-model to determine the configuration for the target day.
    - **Rotational Strategy Application**: Repeat the above steps for each day, treating each as the target day while excluding it from the meta-dataset creation.
    - **Result Compilation**: After the rotation through all days, compile 17 results corresponding to the configurations for each of the 17 days in the dataset.

3.3.9 Scenario9: Model Monitoring

- Objective: The main goal in this scenario is to ensure that drift cases in AI/ML models deployed within an AIF are identified properly, and therefore, take actions regarding model retraining and redeploying in a fully automated way.
- Scenario: The scenario purposed to demonstrate the above objective is to deploy an AIF like the one depicted in Section 2.4.1 but including a specific model and detector, since the one in Section 2.4.1 is described in a more general way. The model used for the validation is a classification model based on a deep neural network, trained with CIFAR10 data. While the drift detector chosen is the Maximum Mean Discrepancy (MMD) with a previous encoder for reducing dimensionality. Additionally, the same CIFAR10 dataset will be prepared but corrupting to the corresponding images in order to change the underlying data distribution. In this case, the method used to corrupt the images is Gaussian Noise. These corrupted images will be used for the detector to detect drift, then retrain the model and re-initialize the detector with these corrupted data, and finally compare the performance of a model that has been retrained with another one that has not.
- Testbed: The tests are carried out at FBK testbed having the NSAP in a minikube instance located in a VM while the MEC system is placed at a separated K8s cluster. The different scenario components are placed in their corresponding infrastructure.
- Metrics: In this validation scenario a single metric is considered, it is the accuracy of the model in inference time. To get this value, it is mandatory to have the ground truth of the inference input data. Although it is known that in a production environment is not possible to know these values, they will be considered for the scenario demonstration.
- Steps:
    - First, an initial bootstrapping step must be done, which consists of: Both training AIFs corresponding to the model and detector should be deployed through the MTO in order to have one version of the model and the detector present on MLFlow/MinIO instance at the NSAP to avoid the MMAIF to fail after deployment. The model and the detector must be trained and initialized respectively using original CIFAR10 data.

- Secondly, The MMAIF should be deployed (also through the MTO) for the model and the detector to be served within the MMAIF, and the corresponding versions continuously monitored by the sidecar.
- Redo the step 2 but in that case without the drift monitoring system, in order to compare the resulting accuracy between both.
- Thereafter, just for validation purposes, a parallel system for collecting the input and output data, calculate and store accuracy, and visualize it is added besides both previously deployed AIFs. This parallel stack will consist of two instances, one MySQL database for the data storage and a Grafana for the accuracy visualization.
- Prepare four different datasets, one with original CIFAR10 data used for training the model and initialize the detector, and another one with the same data but corrupted, in this case will be affected by Gaussian noise. Additionally, prepare two more datasets for test/inference purposes, also original and corrupted. As the CIFAR10 dataset is composed of 60000 images, it will be split into 50000 original images for training and 10000 for test/inference for both cases, original and corrupted.
- Use the original dataset to start the inference over both AIFs, and check continuously the model accuracy, as the model being served is the same by both AIFs, the accuracy should be similar.
- After some time executing the step before, start to make inference with the other dataset containing corrupted images instead of the original.
- The AIF containing the drift monitoring will trigger the retraining of the detector and the model once the drift is detected.
- After the redeployment, it will be seen how the accuracy of the model without monitoring decays while the accuracy of the model monitored by drift keeps a similar level than before.

4 Experimental results

To validate the CCP, the following subsections present the actual measurements derived by executing the methodologies and testing scenarios described in the previous section. The structure is analogous to that of Section 3, i.e., analysing each CCP component/feature separately, as well as altogether in the integrated CCP.

4.1 Scenario1: MTO, MEO, Interfaces & AIF deployment and migration

This scenario will validate the described metrics in Section 3.3.1: (i) Onboarding and deployment time of simultaneous AIFs through the MTO and message broker (Scenario 1.1); (ii) Error rate of AIF's requirements fulfillment (Scenario 1.1, 1.2 & 1.3); (iii) Onboarding and deployment time of simultaneous AIF(s) through the SA MEO (Scenario 1.2); (iv) Full migration for when the MTO is not working (s) (Scenario 1.3); and (v) MEO required resources for the deployment and migration experiments (CPU & RAM) (Scenario 1.2 & 1.3).

The tests were carried out by sending varying numbers of simultaneous instantiation/migration requests: 1, 5, 10, and 15, depending on the specific scenario. For each experiment, 10 iterations were conducted and the mean along with the 95% confidence interval were calculated to estimate the measurements. Figure 46 illustrates the format and content of an individual instantiation request as displayed in the provided Swagger interface for the MTO and the MEO. Moreover, the content of the request via the API appears as follows:

curl -X 'POST' 'http://<IP_MTO/MEO>:8080/meo/aifd/deploy/' -H 'accept: application/json' -H 'Content-Type: application/json' -d '{"aifd_name": "AIF_v4_sentiment_analysis_descriptor.yaml", "namespace": "demo"}'

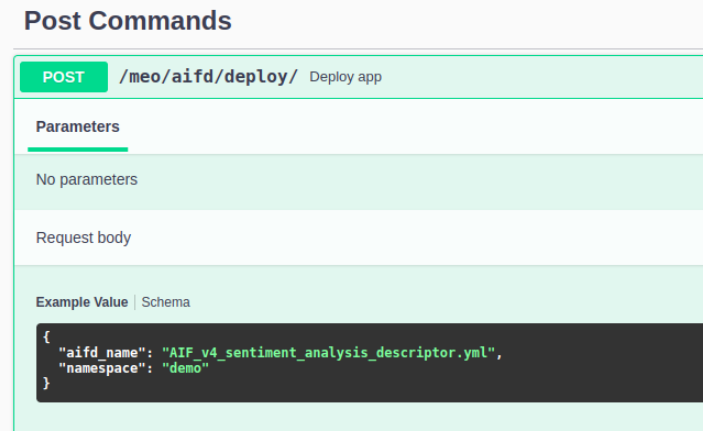


Figure 46 Format and content of an individual instantiation request

The result of a successful request returned by the MTO/MEO includes the unique ID_name of the requested AIF, the repository helm-chart name, the destination namespace, and the destination node. It follows this structure:

INFO 2023-09-13 14:03:21,280 federation.manifold choose_function 57 : mecpm,[{'id': '64f5e414b8ecb0d6bacb43d4', 'name': 'sid-upd-monit-aif-64f5e41211f8d66df7b67cfd', 'bodytosend': {'release_name': 'sid-upd-monit-aif-64f5e41211f8d66df7b67cfd', 'repochart_name': 'aiatedge/sid-upd-monit-aif', 'values': {'nodeSelector': {'kubernetes.io/hostname': 'cluster-worker1'}}}]

- Results of Scenario 1.1:

This scenario displays the average time consumed by the various stages in the deployment process when instantiating from 1 to 15 MEC applications through the MTO. The MTO decides which MEC system and the node for deployment among those fitting the requirements. Figure 47 Notice that most of the time (around 1.25s) is taken by the MECPM helm repo update action. However, the time taken by the end-to-end deployment does not exceed 1.6s even for the highest number of applications.

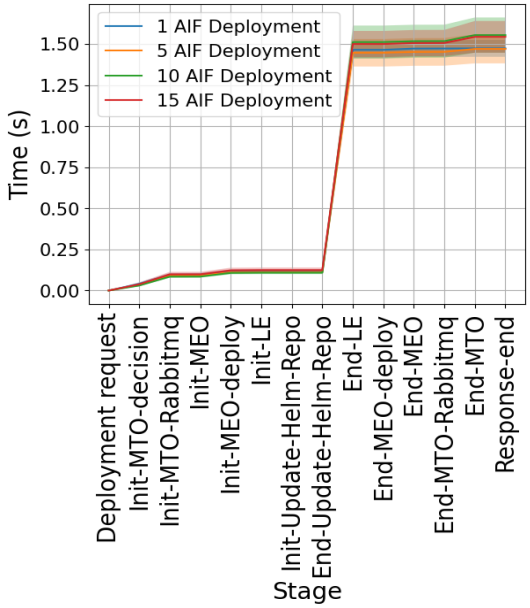


Figure 47 Comparative deployment time for instantiation of simultaneous MEC applications through the MTO

- Results of Scenario 1.2:

This scenario displays the average time consumed by the various stages in the deployment process when instantiating from 1 to 15 MEC applications in one MEC system. The MEO decides the node for deployment among those fitting the requirements. As in the previous scenario, Figure 48 displays that most of the time (around 1.16s) is taken by the MECPM actions. However, the time taken by the MEO's operations does not exceed 0.05s even for the highest number of applications. The entire deployment process, from start to finish, is completed in less than 1.55 seconds, even when dealing with the maximum number of applications.

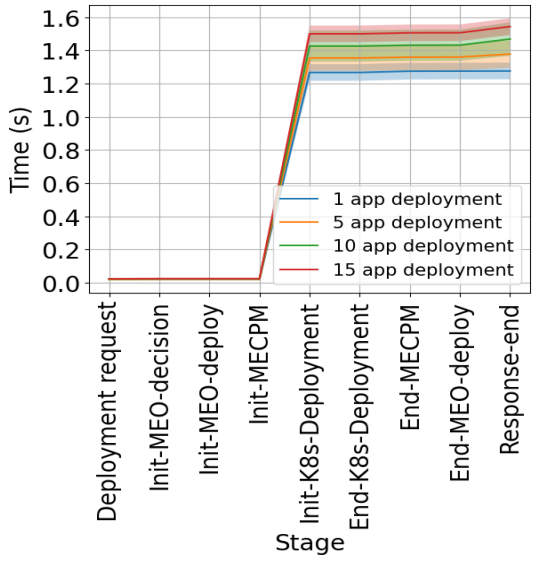


Figure 48 Comparative deployment time for instantiation of simultaneous MEC applications

- Results of Scenario 1.3:

This scenario depicts the migration time across five MEC systems from 1 to 15 MEC applications. As in the previous figure, the MECPM takes the longest time (around 1.16s) is taken by the MECPM actions. However, Figure 49 depicts the time taken by the MEO-to-MEO interface and the MEO's operations do not exceed 0.3s even for the highest number of simultaneous applications. The entire end-to-end migration process is completed in less than 2 seconds, even when dealing with the maximum number of requests.

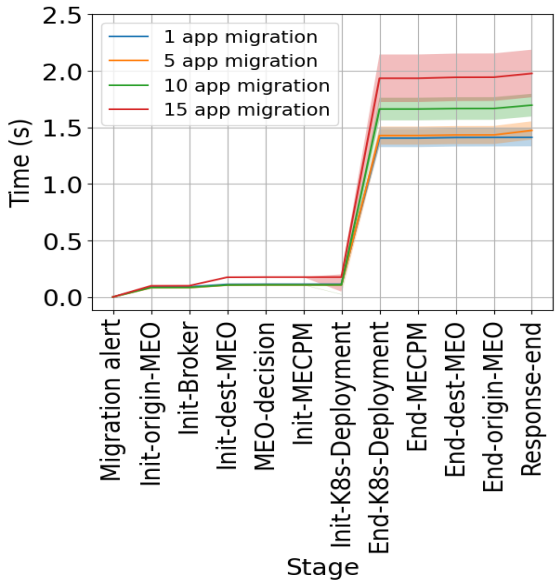


Figure 49 Comparative migration time of simultaneous MEC applications across five MEC orchestrators

Figure 50 showcases the resource utilization of the MEO during the management of instantiation and migration tasks. Given Kubernetes' capability to partition CPU resources down to 1m (10^(−3) Units of CPU), the orchestrator can effectively accomplish the previous tasks with 1 CPU and 100MB of RAM. These results underscore the orchestrator's lightweight characteristic and its ability to perform efficiently.

N. apps	Deployment CPU (Units)	Deployment RAM (MB)	Migration CPU (Units)	Migration RAM (MB)
1	0.19±0.01	88.7±1.0	0.22±0.01	85.5±0.5
5	0.23±0.01	90.7±1.2	0.24±0.02	91.6±0.8
10	0.26±0.02	93.7±1.5	0.27±0.02	91.9±1.2
15	0.30±0.04	95.7±1.7	0.30±0.03	92.3±1.4

Figure 50 MEO resource consumption (CPU and RAM) in simultaneous MEC applications allocation and migration

Figure 51 showcases the error rate of AIF's deployment and migration fulfilment across the different scenarios. The results indicate a remarkably robust performance across all scenarios. In Scenario 1.1, the system consistently achieved a success rate exceeding 94% for various number of simultaneous instantiation requests even with the highest workload. Scenario 1.2 exhibited similar reliability, with success rates surpassing 94%. Scenario 1.3 showed a slightly lower success rate, but still commendable at over 92%.

N. apps	Scenario 1.1 Successes	Scenario 1.2 Successes	Scenario 1.3 Successes
1	10/10	10/10	10/10
5	49/50	49/50	49/50
10	97/100	96/100	94/100
15	138/150	140/150	135/150
Overall	**94,84%**	**95,16%**	**92,90%**

Figure 51 Error rate for simultaneous AIF instantiation and migration requests fulfilment

Concluding Remarks on the Validation of Scenario1

Overall, for MEO and MTO and Interfaces, the above-described sub-scenario results validate the system's robustness and reliability in application deployment and migration tasks. The orchestrators (MTO and MEO) facilitate seamless application lifecycle management and migration across diverse MEC systems, enabled by an innovative federation interface. The experimental results confirm a 95% success rate for instantiation and a 93% success rate for migrating multiple simultaneous requests within federated MEC systems. These findings confirm the system's robustness, efficiency, and reliability in task performance while preserving its modular and lightweight design.

4.2 Scenario2: MPTCP

Let us detail the description of MPTCP-Proxy from Section 2.6. In this section, we provide results about the effectiveness of MPTCP-Proxy. Following validation methodology in Section 3.2.2, tests 1 to 5 will focus on two main functions of MPTCP-proxy: increasing bandwidth and supporting connection when one of the links has problems. Meanwhile, the final test will validate the capabilities of the MPTCP predictive scheduler.

To evaluate the functionality of MPTCP-proxy, we set up a fully isolated local testbed to ensure the experiments are not affected by network interference.

- UE: We use a modem implemented entirely in software from srsRAN. A Wi-Fi kernel module is implemented that allows the UE to have two simultaneous network communications (Wi-Fi and 5G). The bandwidth and latency of each interface are set by the routing device on the path to create different test cases for the UE.
- MPTCP-proxy: Built and packaged for cluster environments, the proxy is quickly deployed as a container on the system. This allows for easy deployment of multiple proxies as well as adjusting the required resources for future proxies as required. As described in Section 2.6, the proxy will be responsible for forwarding data from the connection channel using MPTCP between the UE and the Proxy to the connection channel using TCP between the proxy and the server. This allows connections to existing Internet service servers while still taking advantage of MPTCP.

- Server: We use a webserver application in our tests, to allow UE to download files of different sizes from here.

In the first test, we establish two links of UE connection equivalent in terms of performance with 10Mbps bandwidth. Figure 52 illustrates how throughput can be increased by simultaneously utilizing all available links.

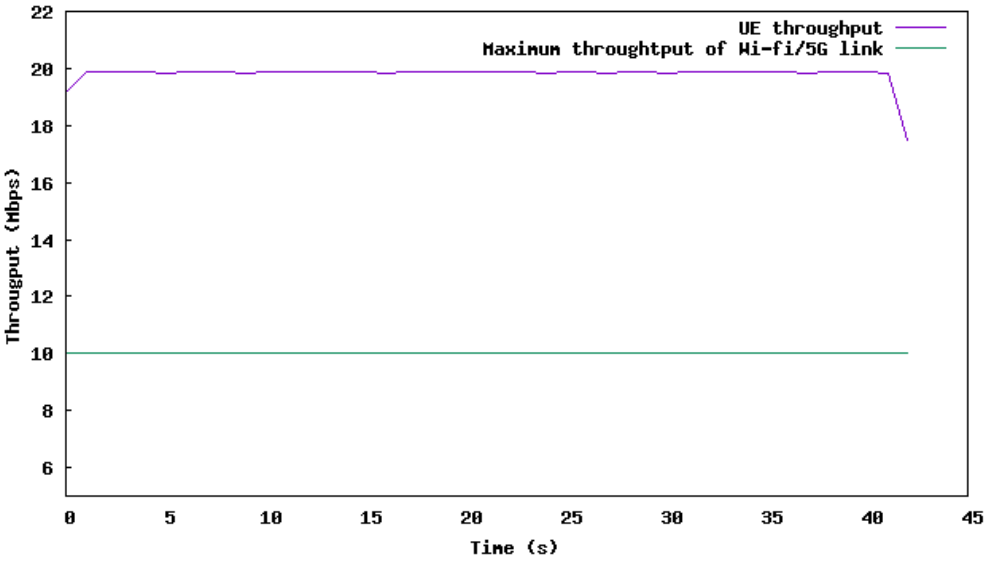
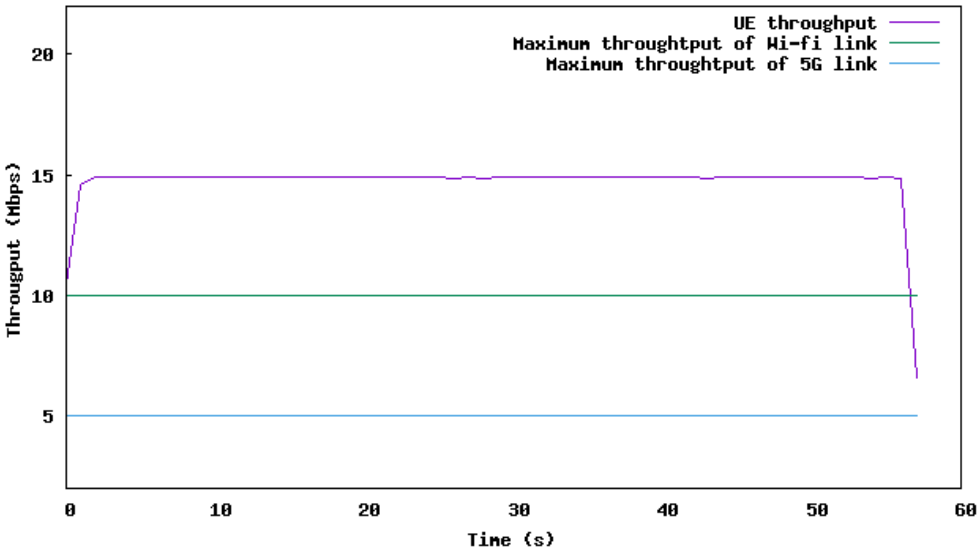
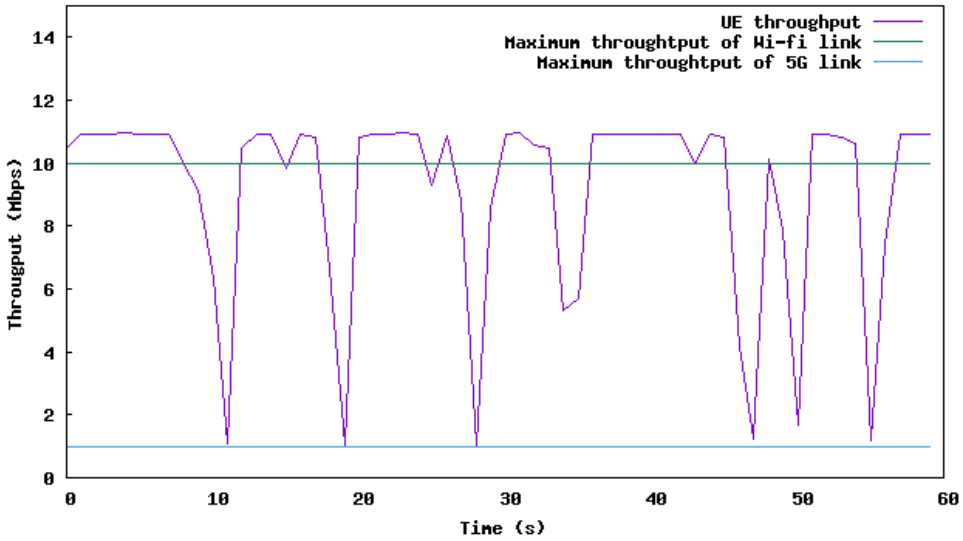


Figure 52 Throughput with two links that are equivalent in term of performance

In the second test, we conducted experiments using to links with varying delays or. Figure 53(a) illustrates that when the difference between the two links is minimal, the UE is still capable of increasing TCP throughput by utilizing all available links at the same time. However, in Figure 53 (b) when the difference is substantial, the throughput becomes unstable due to a Head-of-Line (HoL) blocking problem when the link with lower performance is used. Nevertheless, the average throughput remains equivalent to using only the best connection available to the UE.



(a)



(b)

Figure 53 Throughput with two links that are not equivalent in term of performance

Tests 3 and 4 simulate scenarios in which one of the two links is degraded or completely lost. In test 3, after the first 20 seconds, the bandwidth of the 5G link was degraded to only 1Mbps while the Wi-Fi link remained unchanged. In test 4, the 5G link was entirely lost after the initial 20 seconds. In both cases, the connection between the UE server is not interrupted. However, in test 3 (Figure 54 and

Figure 55), the throughput experiences instability as MPTCP continues to attempt to use a combination of both connections for data transmission. In contrast, in test 4 (Figure 56, Figure 57), MPTCP exclusively utilized the Wi-fi connection.

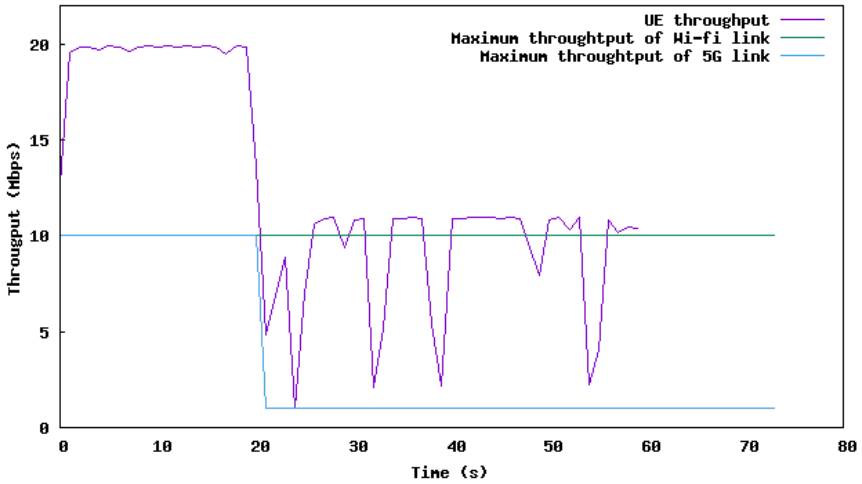


Figure 54 Throughput when one of the two links is degraded.

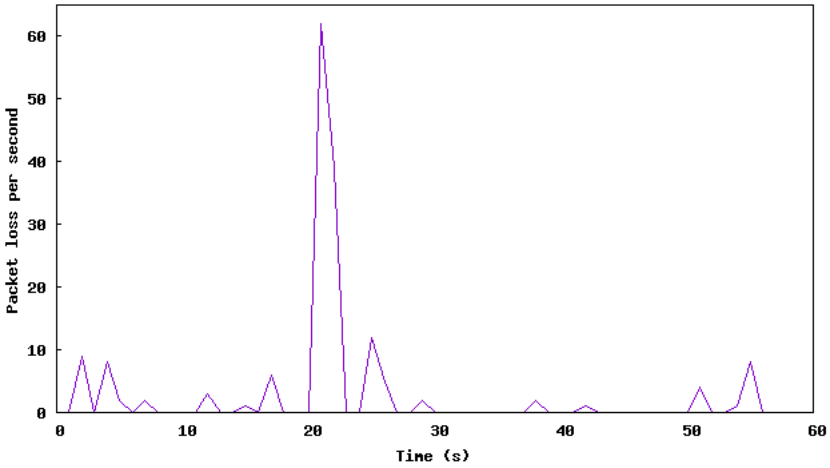


Figure 55 Packet loss when one of the two links is degraded.

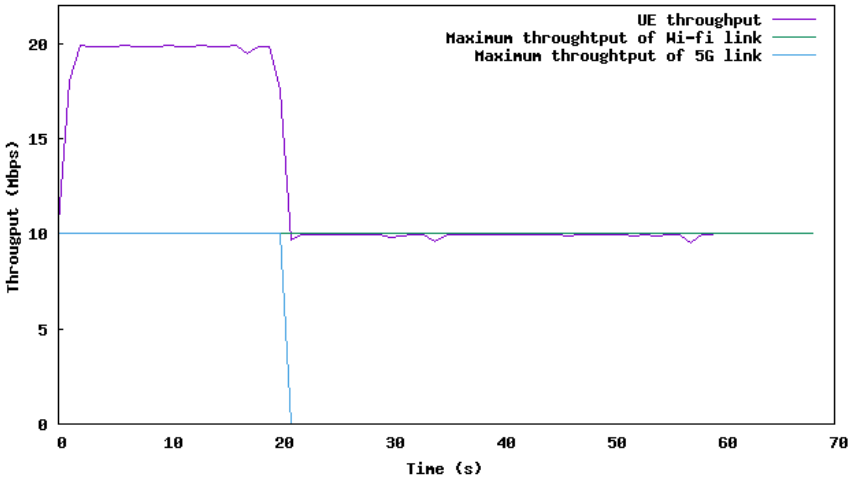


Figure 56 Throughput when one of the two links is lost.

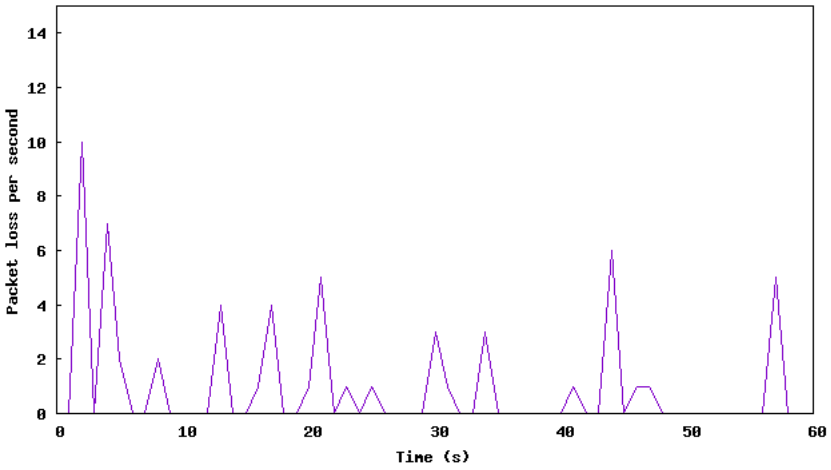


Figure 57 Packet loss when one of the two links is lost

Test 5 (Figure 58, Figure 59) extends the scenario from test 3, where one of the links is restored after a period of failure or degradation. After 20 seconds from the start of connection, the UE's 5G link is temporarily set to a lower bandwidth than its original level. Following 10 seconds of operation at a reduced bandwidth, the link's bandwidth is restored. Throughout this process, the connection between the UE and the server remains uninterrupted, and the UE's throughput is promptly restored to its initial state as soon as the 5G link is back to normal.

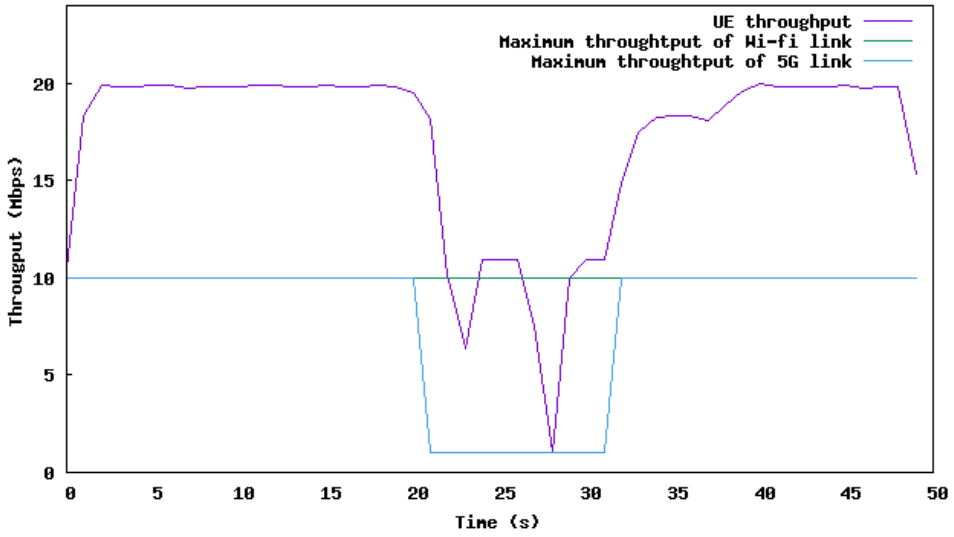


Figure 58 Throughput when one of the links returns to normal after a period of failure or degradation.

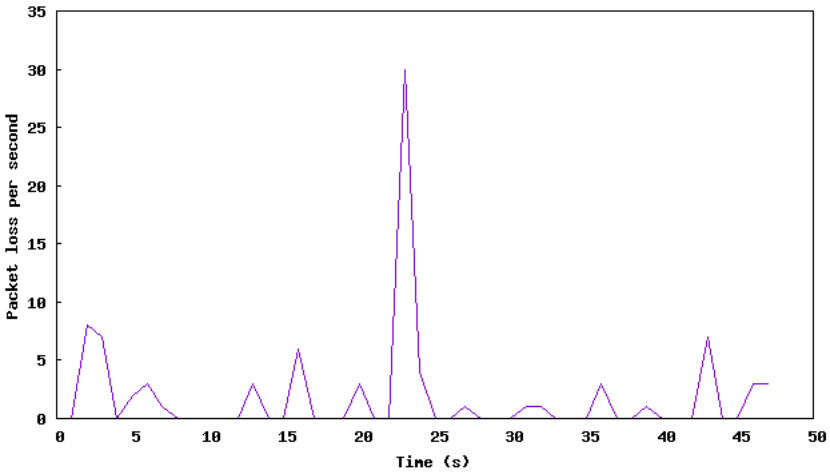


Figure 59 Packet loss when one of the links returns to normal after a period of failure or degradation.

In Test 6, we validated the MPTCP predictive scheduler in the same testbed as the other tests mentioned above. As described in section 2.6.2, we aimed to minimize changes to the MPTCP scheduler's source code. Therefore, as soon as the MPTCP scheduler receives a prediction of packet loss through kernel variables, it temporarily avoids using the path where packet loss is predicted to occur. Once the predicted period of packet loss ends, the kernel variables are reset, and the MPTCP scheduler resumes normal operation.

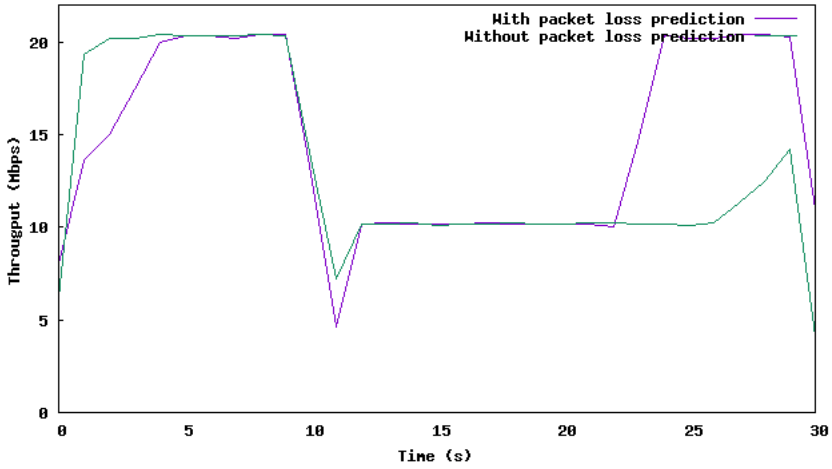


Figure 60 Throughput comparison: With packet loss prediction vs. Without packet loss prediction

Using the same context as the tests above, where the User Equipment (UE) begins downloading a large file from the server, after approximately 10 seconds, packet loss starts occurring in the Wi-Fi connection and continues for 10 seconds. Afterward, the Wi-Fi connection returns to normal. Figure 60 illustrates the throughput in two cases: one with packet loss prediction (Case 1) and one without (Case 2). Since in our test environment, the MPTCP proxy is not far from UE so even in Case 2 (without packet loss prediction), the loss of packets in the Wi-Fi connection is quickly detected, leading to a temporary pause in usage (in a real-world environment with longer propagation delays, the gain is expected to be higher). However, in Case 2, it takes longer to detect that the Wi-Fi connection has returned to normal before resuming usage.

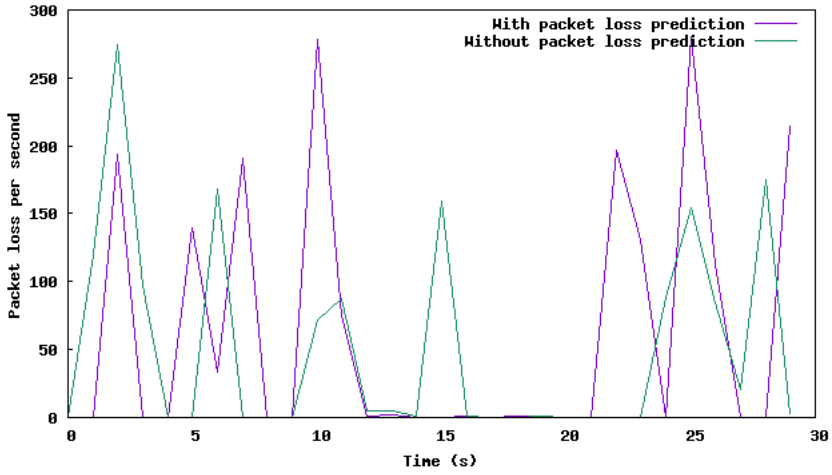


Figure 61 Packet loss comparison: With packet loss prediction vs. Without packet loss prediction

The above Figure 61 represents the number of lost packets. Here, we observe only during the time from the 10th second to the 20th second when packet loss occurs on the Wi-Fi connection. In Case 1, the Wi-Fi connection is not used at all during this period, while in Case 2, the MPTCP scheduler continuously attempts

to use the Wi-Fi connection, resulting in packet loss and an increased number of packets needing retransmission.

Concluding Remarks on the Validation of Scenario2

Overall, as part of the validation of MPTCP in CCP, the conducted tests 1-6 show that we can achieve an increase in TCP throughput across available links in many cases. In our testbed, the UE throughput increases ranges from 50% to 100% with respect to individual throughput reachable by each RAT. Furthermore, the achieved communication reliability is near 100% in the provided PoC upon link failure or degraded channel conditions, and that without acting at the physical nor data-link layer, using an over-the-top (transport layer) approach. The MPTCP-interface state prediction at the xApp does enhance the multipath scheduler performance with respect to normal operations, anticipating packet loss hence maintaining connection performance. As future work, further work would be needed to test these advancements in real conditions. Moreover, integration of the proxy function within the UPF system is another desirable future work in the area.

4.3 Scenario3: Acceleration

By following the validation methodology described in Section 0 for the acceleration testbed of Section 0, and the development steps 1-4 already performed for certain AIFs (Section 2.7), we generate our acceleration results described below. We compare Connect-Compute Platform.

First, we focus on the acceleration of various CNN-based AIFs on FPGA and GPU. For the former, we utilize Xilinx U280 (near-edge) and Xilinx ZCU104 (far-edge), while we implement and containerize the AIFs over Vitis-AI (as described in Section 0). For the latter, we utilize the Nvidia V100S Tesla GPU (near-edge) and Xavier AGX (far-edge) by employing the TensorRT framework. For comparison, we also execute the containers on an Intel Xeon(R) Silver 4210 CPU (near-edge) by employing the TFLite framework.

Table 4 Execution time (latency in msec) of 5 representative CNN AIFs in far- and near-edge devices (blue and green lines)

Device / AIF	LeNet-5	MobileNet-V1	ResNet-50	Inception-V4	ResNet-152
FPGA ZCU104	0.200	2.676	11.442	33.731	58.548
GPU Xavier AGX	2.464	8.713	14.823	X	X
CPU ARM-A53	1.413	76.543	380.215	1070.451	1834.387
FPGA Alveo U280	0.217	1.141	12.745	19.464	46.945
GPU V100S	0.445	1.051	2.275	6.663	4.100
CPU Xeon S. 4210	0.393	14.623	63.731	164.348	258.137

The CNN results above show significant acceleration factors vs the native CPU, especially when considering far-edge devices and larger CNN networks (top-right quarter of Table 4). At the near-edge, the GPU provides better acceleration compared to the FPGA programmed with Vitis-AI (i.e., the soft-core DPU), especially for the large AIFs. When comparing the far- to near-edge devices, we get almost one order of magnitude faster execution when considering CPU/GPU, but not when comparing FPGAs (attributed also to the utilized soft-core DPU). Overall, when we have an AIF at far-edge, we can achieve

one order of magnitude faster execution by migrating to the near-edge CPU (3-7x improvement), or by executing it on an accelerator at the far-edge (which achieves even 2-5x faster execution than the near-edge CPU). At extreme cases, the CNN latency improves by up to almost 500x (when migrating a very large AIF from a far-edge CPU to a near-edge GPU).

Our second benchmark was the LSTM AIF (with network intrusion detection as example dataset). This AIF uses a 3-layer LSTM model with an encoder, a decoder, and a dense layer. The model has 128 units for the encoder layer, 128 units for the decoder layer, one input/output feature and requires 50 time-steps to complete the process. For the recurrent activation uses a sigmoid function and for the activation a tanh function. The input vector has a size of time steps*features elements. We explored implementations with floating- and fixed-point arithmetic and a wide memory interface (512bit). Table 5 and Table 6 present the resource utilization percentages and latency considering implementations with various optimizations on the Alveo U280 device and next on the MPSoC ZCU104. It's worth noticing that the floating-point (FP) implementation requires much more resources and latency than the 16bit fixed-point for both devices. Also, from experiments, 16bit arithmetic provided a good tradeoff between accuracy, resources, and performance thus we kept this precision. Last, for this type of LSTM, the smaller FPGA (ZCU104) and the larger U280 FPGA achieve similar performance (the same circuit/design fits in both devices). The latency, however, is not the same when using large batch size on the bigger device, especially combined with 512-bit interface, thus we achieve better time/sample with 128 batch input on U280. Comparing latency (batch=1) results that of the far-edge CPU (i.e., ARM-A53 with TFLite), the far-edge FPGA achieves 25x acceleration, whereas compared to the near-edge CPU (i.e., Intel Xeon(R) Silver 4210), the near-edge FPGA achieves 5x acceleration (the near-edge CPU is 7x faster than the far-edge CPU).

Table 5 FPGA resources and execution time of a representative LSTM kernel on a near-edge FPGA (Xilinx Alveo U280)

U280 resources	LUT 1304K	LUT Mem 590K	REG 2607K	BRAM 2016	DSP 9024	Time
FP	39.07%	25.25%	26.66%	2.26%	57.55%	3.38ms
16bit fixed (float I/O)	10.63%	11.88%	4.36%	1.10%	12.18%	2.14ms
16bit fixed (fixed I/O)	10.79%	11.87%	4.67%	2.26%	12.18%	1.20ms
16bit fixed (fixed I/O, 512bit interface)	15.05%	6.38%	6.49%	2.26%	16.44%	0.72ms
16bit fixed (fixed I/O, 512bit interface, x2 engines)	21.47%	6.43%	9.08%	2.43%	28.60%	0.66ms
16bit fixed (fixed I/O, 512bit interface, 32 batch input, 4x LSTMs)	35.49%	24.66%	12.74%	3.58%	52.99%	7.31/32 = 0.23ms
16bit fixed (fixed I/O, 512bit interface, 64 batch input, 4x LSTMs)	35.49%	24.66%	12.74%	3.58%	52.99%	14.04/64 = 0.22ms
16bit fixed (fixed I/O, 512bit interface, 128 batch input, 4x LSTMs)	35.49%	24.66%	12.74%	3.58%	52.99%	27.37/12 = 0.21ms

Table 6 FPGA resources and execution time of a representative LSTM kernel on a far-edge FPGA (Xilinx Zynq MPSOC ZCU104)

	LUT	LUT Mem	REG	BRAM	DSP	Time
ZCU104 resources	230K	102K	460K	624	1728	
16bit fixed (float I/O)	37.50%	36.57%	14.63%	17.67%	63.60%	1.41ms
16bit fixed (fixed I/O)	56.21%	69.03%	22.71%	17.67%	63.60%	1.18ms

By following the above-described approach, we continued benchmarking various inference AIFs on various HW nodes. In each case, we used HLS coding or high-level vendor-specific tools for implementation (e.g., Xilinx Vitis AI), together with optimization exploration of basic code parameters (e.g., internal word-lengths and arithmetic types in the AIF, most suitable to the underlying HW). We created a representative set of inference benchmarks and nodes, as summarized in the following table.

Table 7 benchmarking of AIFs in acceleration cluster

AI benchmarks (inference)	**HW nodes**
- **multiple LSTM** auto-encoders (2-4 layers, 2-50 timesteps, 1-230 features) i.e., total parameters 0.07-1.1M - **multiple MLP** networks (1-5 layers, 1-10K neurons, 10-2000 inputs), i.e., total MOPS 0.01-20 - **multiple CNN** for classification, segmentation, and object detection (Lenet, small MNIST and CIFAR28 networks, custom ``shipdetection'', Mobilenet v1.0/3.0, Resnet 50 v1, Resnet 152 v1/2, Inception v4, SSD MobileNet v1), I.e., total FLOPS from 1MFLOP to 42GFLOP	Nvidia V100 GPU (near-edge), Xilinx U280 FPGA (near-edge), Intel Xeon CPU (near-edge), Nvdia Jetson Nano and AGX Xavier GPU (far-edge), Xilinx ZCO104 MPSOC and Versal ACAP FPGA (far-edge), ARM Cortex A53 CPU (far-edge)

We grouped the results based on AI type and workload (low, medium, or high, according to the FLOP performed by representative networks in each category). Furthermore, we distinguished our results based on the goal of each AIF, i.e., the latency of single-shot AIFs or the throughput of batch-processing requests (each goal affects the code generated for the AIF). Table 8 summarizes these results per node type (single-thread or multi-core CPU, FPGA, GPU) and edge site (far or near, i.e., server or embedded HW devices).

Table 8 Benchmarking results of acceleration in far and near edge

INFERENCE			NEAR-EDGE (server)				FAR-EDGE (embedded)				speedup (near)		speedup (far)	
type	workload	goal	CPU-1	CPU-N	GPU	FPGA	CPU-N	GPU	FPGA	TPU	GPU	FPGA	GPU	FPGA
MLP	low	latency	<0.1ms		0.3ms	0.3ms	<1ms	2ms?	.4-0.5ms	0.3ms	0	0	0	1
		thrghput		20-24K	30M	6-8K	1-2K?		4-6K		...	0		4
	mid	latency	1.3ms		0.3ms	1ms	6ms		1.5ms	0.3ms	4	1		4
		thrghput		14K	6M	5.5K	0.2-0.8K		1.5K		430	0		4
	high	latency	6ms		0.4ms	1ms	26ms		>2ms		15	6		10
		thrghput		9K	2M	5K	38		1K		222	2		20
CNN	low	latency		0.5-1ms	0.5-1ms	.2-0.4ms	1-2ms	1.4ms	.1-0.3ms	0.5ms	1	2	1	10
		thrghput			15K	5K	4K		5-6K	2K				1
	mid	latency	40-250m	10-65ms	1-2ms	1-13ms	20-400m	5ms	1-12ms	0.6ms	14-28	5-14	4-80	17-35
		thrghput	5-25	20-80	6-20K	0.25-1K	2-10		150-700	1667	250-300	12		10-70
	high	latency	0.9-1.5s	.16-.26s	4-7ms	19-47m	1-2s	91ms?	30-58m	103ms	25-65	6-9	18	32-37
		thrghput	0.5-1	5-8	1.5-2.5K	50-220	0.5-1		35-60	10	300	10-28		70
RNN	low	latency	0.3ms	0.3ms	1ms	0.3ms	2ms	1ms	0.3ms		0	1	2	4
		thrghput	3000	30K**	60K**	3K	0.5K	1-5K	2K		2	0	4-8	4
	mid	latency	18ms	1.5-3ms	2ms	0.8ms	5-30ms	1ms	0.4-1ms	2-4ms	1	2	4-30	4-45
		thrghput	55	5K	30K**	4K*	30-200	1-5K	1-2K	0.5K	6	1	4-90	4-45
	high	latency	50ms	11-20ms	13ms	5ms	100ms	10ms	5ms		1	4	10	20
		thrghput	20	0.5K	5K**	0.6K*	10	100-500	200		10	1	10-50	20

The results show that relatively small AIFs, such as RNN or MLP executed in less than a millisecond per input on a near-edge CPU, are not worth accelerating (e.g., due to the overheads of initiating the HW accelerator being comparable to the AIF's CPU time). High-end GPU HW is prevalent for accelerating MLP+CNN at the near edge, while embedded FPGAs are better for low-power and latency-critical applications at the far edge. The FPGA is preferable when accelerating LSTM in latency-critical applications and assuming optimizations via HLS. Furthermore, migrating an AIF from a far-edge node to a GPU server might not be worthy for many CNN latency-critical apps due to added link latency.

Overall, Table 8 shows that the heterogeneity of HW accelerators and AI characteristics plays an important role when selecting the placement of an incoming AIF to the cluster, even when the cluster is idle or fully available to the user (the problem becomes even more complicated when dealing with prior cluster utilization and multi-tenancy scenarios). Regarding device acceleration factors (last columns in the table), the gain lies in the broader area of 10x for latency and 100x for throughput, with accelerators becoming even more important at the far-edge.

Extended testing of AIF acceleration on GPU cluster

In order to further evaluate performance of different types of GPUs that can be used at the near and far edge sites we implemented an additional testbed at Italtel premises as described in Figure 62 and Figure 63.

The equipment related to an Edge environment includes:

1. 2 Jetson TX1 (acting as workers in a Kubernetes cluster, a VM acts as master).
2. 1 Jetson XAVIER (stand-alone).
3. portable server (2 x Intel® Xeon® Silver 4114, 10 Core) hosting NVidia P4 on the PCIe bus – running Kubernetes (cf. Tesla-P4-Product-Brief.pdf, nvidia.com).
4. HP workstation for hosting CCP/NSAP.

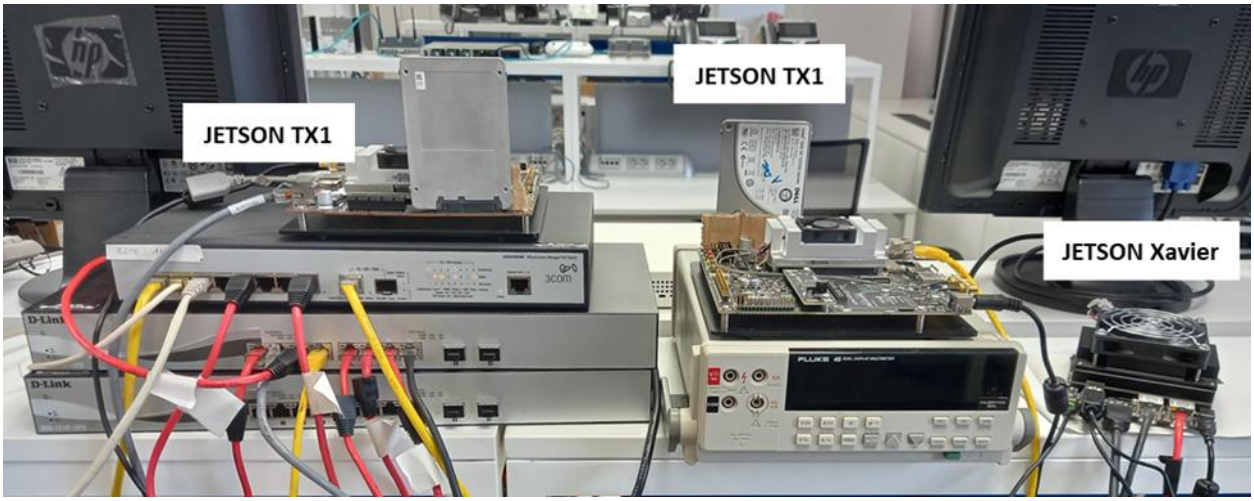


Figure 62 Italtel Far Edge system

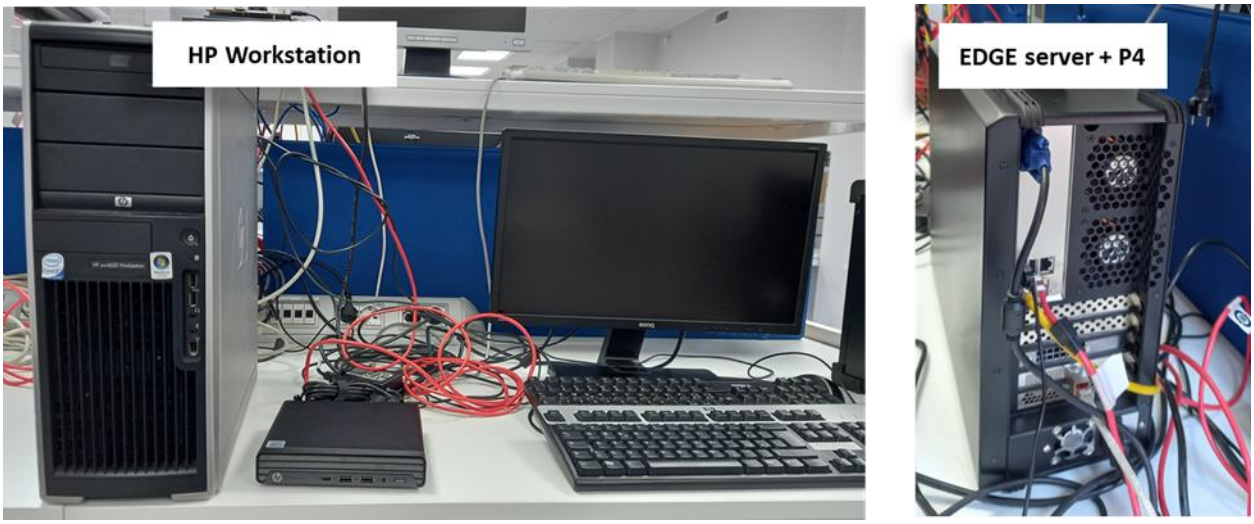


Figure 63 Italtel near edge system

The first step was to identify AIFs, mainly related to video analysis, to be used for the evaluation purposes and test the training and inference process.

We selected an AIFs related to a “Women Safety” use case. The use case aims to first recognize a woman walking in a smart city and then to analyse the type of behaviour of the people around her, to verify for a possible act of violence. The technology used is the following:

- Person detection & tracking
    - YOLO6(c.f. https://github.com/meituan/YOLOv6)
    - DeepSORT(c.f. https://github.com/nwojke/deep_sort)

- Action recognition:
    - CNN+RNN
    - Transformer (hybrid CNN vs ViVit)
    - Google MoViNet
    - VideoMAE
- Datasets:
    - GitHub - mchengny/RWF2000-Video-Database-for-Violence-Detection: A large scale video database for violence detection, which has 2,000 video clips containing violent or non-violent behaviours.
    - GitHub - airtlab/A-Dataset-for-Automatic-Violence-Detection-in-Videos

Considering the obtained results, reported in Table 9, the far edge cluster based on Jetson TX1 is not capable to run the training phase (due to lack of memory), the Jetson Xavier AGX is performing better but it is not capable to run the training process. Therefore, the training must be done at the near edge, and the inference at the far edge on the P4 based system.

Table 9 Italtel Results

	Action recognition	Tracking	FPS
P4	Inference in 36 ms, avg 23 ms	Inference time: 33 ms	19.3
XAVIER	Inference in 52 ms, avg 56 ms	no	8.9

Automation Framework (TF2AIF)

Regarding our developed framework (TF2AIF) that automates the conversion of Tensorflow2 models into fully-fledged AIF-container implementations across multiple platforms (adhering to the AIF and CCP specifications), the results show automation significantly reducing both the development time and the validation efforts required. The automation process for building containers, combined with automatic code generation, alleviates a significant portion of the developer's workload. Moreover, the speed of validating the resultant containers is enhanced due to the shared inputs among them. By verifying the proper functioning of just one container, it is ensured that the entire AIF operates as intended. The framework effectively shrinks the engineering time from a day to some 20 minutes, thus allowing rapid realization of AIFs.

Our AIF-container implementations tap into various inference acceleration frameworks for optimal performance across different hardware.

Table 10 AIF implementation inference acceleration frameworks

Platform	Inference Acceleration Framework
AGX	ONNX runtime with TensorRT backend (INT8)
ALVEO	Vitis AI (INT8)
ARM	Tensorflow Lite (INT8)
CPU	Tensorflow Lite (FP32)
GPU	Tensorflow-TensorRT or ONNX runtime with TensorRT backend (FP32/FP16/INT8)

The AIF-container implementations follow the monitoring scheme specification.

In more detail, the framework requires:

1. AI Model: A model conceptualized and created using the Tensorflow2 framework.
2. Server-Side Code: Approximately 100 lines of code to dictate input decoding, data preprocessing, data postprocessing, output encoding, and other experiment-specific operations.
3. Configuration Files: Essential details pertaining to the AIF, such as batch size and server IP.
4. Dataset: Data that the client will send to the server, in any form.
5. Client-Side Code: Approximately 100 lines of code, specifying data transfer between the client and the server using the REST API.

In the beginning, the framework converts the Tensorflow2 model to the appropriate form to be used by the different inference acceleration frameworks we use for each platform. Afterwards, it combines the user-provided code with auto-generated code, accounting for individual framework nuances and limitations, to realize a server environment for each platform. Finally, based on the user provided configurations, a working AIF container is created for each platform and uploaded to the specified repository, ready for deployment to the CCP. One additional feature is the creation of an example client container that is designed to appropriately interact with the AIF servers.

IARM, proof-of-concept AI evaluation

As described in Section 2.7, one of the key sub-components that can be introduced in the resource management of the compute nodes, i.e., in IARM, can be Deep Reinforcement Learning (DRL). Initially, the DRL approach can be introduced to handle dynamically AIF workloads on CPU cores within a MEC system, while preserving the performance of the accelerators. We proceeded into a preliminary development and evaluation of a DRL agent that is responsible for deciding the migration of AIFs (from one CPU node to another) and managing the amount of CPU resources assigned to a specific AIF instance that are available for leverage during its execution.

The DRL agent is engineered towards successfully serving a user defined QoS constraint while striving to minimize resource utilization and interference applied to accelerators in a multi-tenant setup. First, we investigate whether a target QoS is achievable from at least one placement configuration, and afterwards we look for more candidate solutions utilizing less resources while trying to avoid undermining the performance of co-hosted accelerators. In a dynamically evolving environment, it is quite common for a

cluster node to get stressed from multiple requests and sources, and thus possible migrations of workloads to less stressed nodes should be considered for ensuring the end user QoS. A DRL agent, by employing a Deep Q-Network, learns to map complex cluster states to scheduling decisions that are not easily foreseen from heuristic approaches due to high-dimensionality of input vectors and hidden high-level correlations among them.

In our experiments, we developed a DRL agent via the Stable Baselines 3 library, that offers multiple implementations of RL algorithms, and evaluated its performance within OpenAI Gym's environment. We leveraged the DQN algorithm that builds on combing an off-policy, Q-learning approach with a neural network responsible for calculating the Q-values for (state, action) pairs. The calibration of the agent's training is done through a reward function, e.g.,

$$Reward = \begin{cases} K - util, & latency \leq QoS \\ -|\frac{latency - QoS}{QoS} - util|, & latency > QoS \end{cases}$$

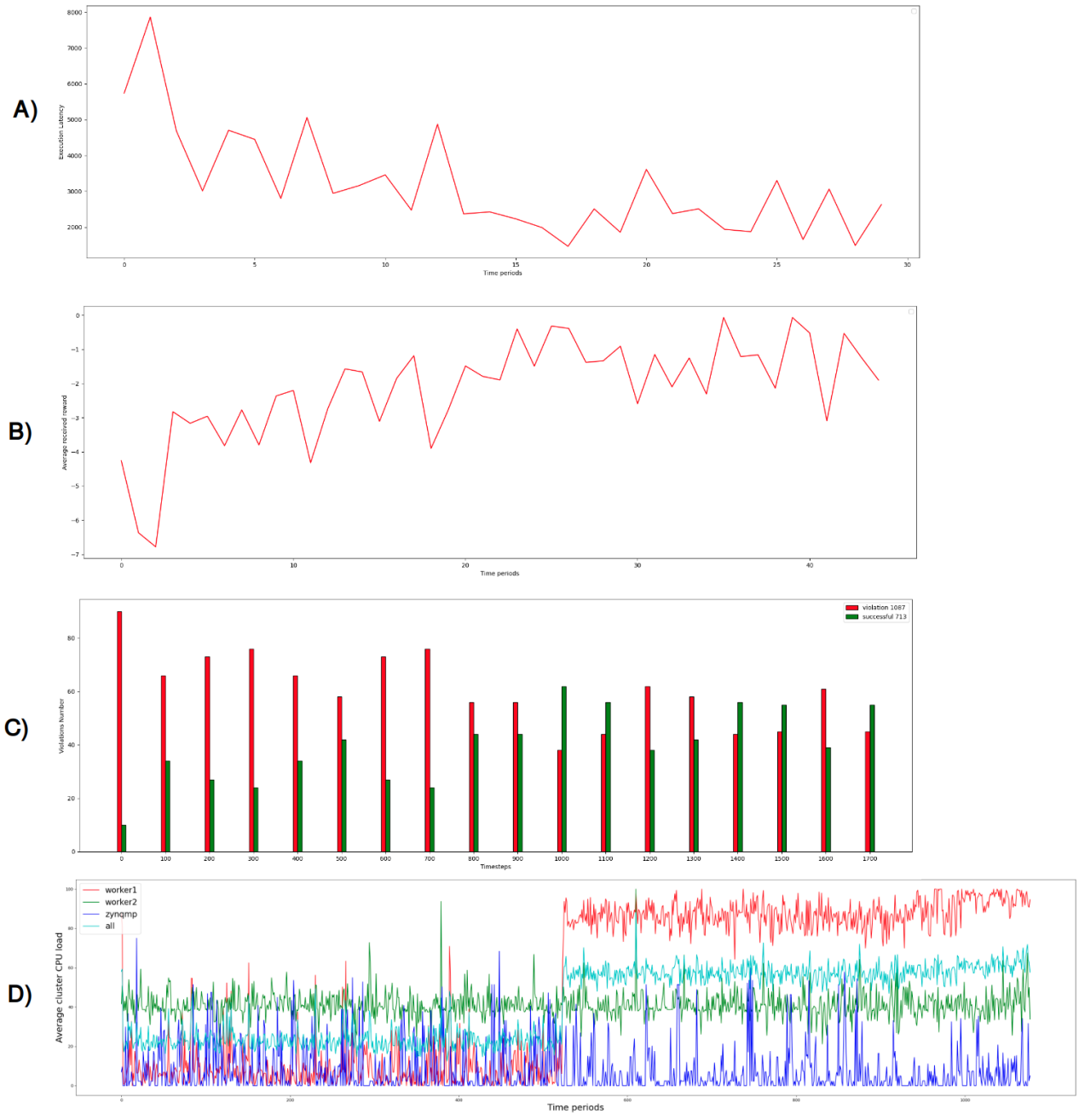


Figure 64 proof-of-concept of example RL executed inside IARM for managing CPU resources.

The figures show the RL performance during 1800 steps of training. A) Execution latency vs time, B) Average received reward over time, C) QoS violations (red bars) over time, D) example CPU workload over time.

During our RL experiments, we set a discrete level of QoS that is achievable only by a small subset of AIF topologies and thus is not trivial to be solved. After the agent converges to an accepted solution, we apply interference and increase the load within some cluster nodes. In this way, we force the agent to change its decisions based on the new circumstances and test whether it finds other acceptable solutions for the load given at that moment. Eventually, the agent manages to serve the QoS with different AIF placement configurations than the ones decided in the first place. The plots in Figure 59, above, show the agents' convergence during training but also the RL's monitored performance during inference.

Concluding Remarks on the Validation of Scenario3

Overall, the evaluation of the acceleration parts in CCP has shown that there exists great potential to improve the compute capability across the Cloud-Edge continuum, by an order of magnitude, compared to the straightforward CPU-only solution. In the testbed, the accelerators themselves already provide such a huge boost, individually, in far and/or near edge comparisons. In the CCP, the proposed IARM structure can support a bigger orchestration scheme that is aware of the acceleration resources and can assign the correct SW to the most suitable HW nodes. The intelligence inside IARM for optimizing the resource management is still a world-wide open research area and our first proof-of-concept implementations have shown that the proposed resource management approach is feasible. Furthermore, automated frameworks for deploying AIFs to very heterogeneous HW was also shown to be feasible (TF2AIF), i.e., an integration of tools from different vendors towards improving the usability of CCP in what concerns its acceleration resources.

4.4 Scenario4: Serverless Platform

By following the methodology described in Section 3.3.4, we propose the results of the validation process we produced regarding the usage of Serverless Platform. Here follows a description of the steps we took to orchestrate the process, the measurement, and metrics we collected with an analysis of the results and a 'lesson-learned' section regarding the possible application scenario in the AI@EDGE perspective.

Procedure:

- Testbed preparation: We deployed all the necessary components to deploy an AIF function. We created a dedicated namespace for the deployments, deployed the CCP with Prometheus, LightEdge, and the Stand-alone version of the MEO, deployed SeldonCore, an MLFlow Server, and a MinIO Storage.
- Saving the Model: We saved a generic machine learning model built with Scikit Learn in MLFlow format on a MinIO storage system using a Python script. This ensured proper version control and easy access to the model for deployment. This is also a requirement for the Seldon Deployment which fetches the ML model from remote storage during the initialization of the deployment.
- Building the Docker Image: To measure the time used to build a Docker image, we created a Dockerfile that copied the ML model into the image directly (in other words, the model was hardcoded in the image with all the dependencies it needed). We recorded the time taken to build the image, including dependencies and configurations.

- Creating Deployment Plans with Helm Charts: We generated two different deployment plans using Helm charts. One plan was for deploying the manually built Docker image, and the other for the Seldon deployment. Helm charts helped us encapsulate configurations, simplified, and parameterized the deployment process. We uploaded the Helm Charts on the helm-chart-repository hosted at GitLab.
- AIF Descriptor: We created an Artificial Intelligent Function (AIF) descriptor for both deployment solutions (manually built image and Seldon deployment). The descriptor contained essential information about the settings required for the AIF services.
- Deployment with MEO: We deployed the AIF services using a standalone MEO version on the FBK testbed. The deployment took place in the dedicated namespace.
- Measuring Deployment Time: We measured the time it took for the deployment to be ready by using a Python script to check the status of the deployed pods. This allowed us to understand how long it took for the AI function to be fully available for use. We measured the time delta between the scheduling and the readiness of the pods.

Metrics and analysis

Image building

We took Docker image building time in two case scenarios. The first one is the case of a building from the ground up. Here follows a simplified terminal output where we retrieved the building time metric:

```
[+] Building 33.0s (8/8) FINISHED                    docker:default
 => [1/3] FROM ghcr.io/mlflow/mlflow:v2.5.0          14.4s
 => [2/3] RUN pip install mlserver-mlflow            16.7s
 => [3/3] COPY ./model /mnt/models/model              0.0s
 => exporting to image                                0.7s
 => => naming to docker.io/library/test
```

Figure 65 Docker image building time from the ground up

As noted, we opted for a base MLFlow image to build the container due to the utilization of an ML model saved with MLFlow during testing. To replicate the functionality of Seldon Deployment, we installed the mlserver-mlflow Python package on the image. Seldon uses the MLServer package as a tool to expose the model as a service. Another advantage of using MLServer is that it already includes Scikit Learn as a dependency, which spared us from installing additional dependencies and contributed to a resource-efficient profile.

The following screens illustrate the same procedure, but this time it only overwrites an already existing model. This scenario is common when a model needs to be updated. Moreover, the image is cached, so, in this case, there was no need to download the image from a remote repository.

```
[+] Building 0.8s (8/8) FINISHED                    docker:default
 => [1/3] FROM ghcr.io/mlflow/mlflow:v2.5.0                  0.0s
 => CACHED [2/3] RUN pip install mlserver-mlflow             0.0s
 => CACHED [3/3] COPY ./model /mnt/models/model              0.0s
 => exporting to image                                       0.0s
 => => naming to docker.io/library/test                      0.0s
```

Figure 66 Docker image building time from image already existent

From a cached image and with a small model, it took a small time to overwrite a previous model.

In synthesis:

Table 11 Image building times in seconds

Building mode	Time (sec)
Building time from ground up:	33.0
Building time with model overwrite and image caching	0.8

An additional consideration should be taken into account concerning the measurements of image building time. We conducted the image build on a local machine with a fast internet connection and minimal dependencies requirements for running the ML model. However, in different scenarios, such as when a model necessitates GPU libraries like TensorFlow or PyTorch with CUDA, or when dealing with slower internet connections, it is expected that the image building process might take more time.

Deployment time

We performed the measurements of deployment time through a Python script that connects to the Kubernetes APIs and retrieves the deployment times for each pod. The measurement is based on the metrics of the pods' PodScheduled and Ready states. We executed the script for both deployments, obtaining the following results:

Table 12 Building time measurement for Raw-build and Seldon-build deployments

	Raw-build deployment (sec)	Seldon-build deployment (sec)
	2.00	26.00
	2.00	25.00
	1.00	26.00
	2.00	25.00
	1.00	26.00
	2.00	25.00
	2.00	25.00
	1.00	26.00
	2.00	25.00
	1.00	26.00

Mean time	**1.60**	**25.50**

As we see, the prebuilt image (raw-build) deployment took a mean time of 1.6 seconds in 10 deployments we tested. On the other hand, the seldon deployment took 25.5 seconds. The reason for the overhead of the seldon-build is related to the fact that every Seldon Deployment starts with an InitContainer that fetches a model from a remote storage, and only after that Seldon boots the container that loads the model as a service. Practically, this can be translated as two containers initialized in the Seldon pod, when in the case of the raw-build it's only one.

Memory space usage

A final measurement we took is the amount of disk space used by the respective deployments. The raw building image was uploaded on the AI@EDGE docker registry, while the image used by Seldon is hosted on docker-hub. The size image metrics are these:

Table 13 Image sizes in MB

Image	Size (MB)
registry.gitlab.com/aiatedge/helm-chart-registry/raw-build:v1	928
seldonio/mlserver:1.2.3-mlflow	1770

Concluding Remarks on the Validation of Scenario4

Overall, as observed from the results obtained by measuring various aspects of the deployment lifecycle, there is a substantial time resource investment in creating a Docker image from scratch. Furthermore, two factors should be considered: the time taken to upload the image to Docker Hub (or other repository) and the manual image building process. Currently, CCP does not support automated creation of AIF images, necessitating potential external implementation of the process. Regarding overhead times, if the scenario requires a fast deployment, a pre-built image might be the better option in terms of timelines. However, in cases like those presented in WP3 of anomaly detection, where data drift is a significant concern, Seldon deployment proves more effective as it allows for reloading a model without rebuilding the image. In any case, it has been also validated that even custom resources such as Seldon deployments can be easily deployed through CCP. Following the approach presented in Section 2.2, the use of the Serverless frameworks allows for completely declarative deployment approach, where the AIF descriptor defines only the type of the execution framework and model reference, while the implementation relies on the standard tools and patterns.

4.5 Scenario5: Integrated CCP

This scenario intends to validate the integration of the IOC and the IARM, achieving a complete integration for placing AIFs. In these scenarios, the IOC will receive aggregated metrics from the CCP, to take placement decisions at a MEC system level, while the IARM will oversee selecting the optimal node for each AIF.

Test1:

We deployed a single CPU version AIF in the FBK CPU worker node, performing either a segmentation or a classification task. The AIF deployment was done a) without MEO or IARM, b) with MEO only, and c) with MEO and IARM. The following tables show the overhead \ wait time until each deployment and whether the deployment was completed successfully. Numbers vary depending on the time required to pass data between the CCP components and the time required for each component to complete its task.

Table 14 Integrated CCP Test 1 results

Deployment	**AIF**	**Task**	**Overhead (ms)**			
			Mean	**Median**	**90th percentile**	**Correctness**
without IARM-MEO	CPU	segmentation	241.3	244.5	390.7	yes
	CPU	classification	198.9	175.5	331.0	yes
with IARM	CPU	segmentation	1013.7	998.0	1299.4	yes
	CPU	classification	1074.6	1089.5	1441.6	yes
with IARM+MEO	CPU	segmentation	1248.1	1253.0	1610.8	yes
	CPU	classification	1336.4	1283.0	1687.6	yes

Test2:

A. We deployed a CPU only version AIF performing either a segmentation or a classification task, and an FPGA version AIF performing a segmentation task, in the ICCS remote accelerator node.

Table 15 Integrated CCP Test 2A results

		ICCS accelerator node			
AIF	**Task**	**Overhead (ms)**			
		Mean	**Median**	**90th percentile**	**Correctness**
FPGA	segmentation	4554.9	3885.5	5740.3	yes
CPU	segmentation	4242.3	3818.0	4739.1	yes
	classification	4310.0	3867.0	4786.0	yes

B. We deployed for segmentation and classification tasks:

- an FPGA AIF in the remote ICCS accelerator node
- a CPU AIF in the remote ICCS accelerator node
- a CPU AIF in the FBK CPU worker node

Table 16 Integrated CCP Test 2B results

Classification							
Node	**AIF**	**E2E Latency (ms)**			**Throughput (fps)**		
		Mean	**Median**	**90th percentile**	**Mean**	**Median**	**90th percentile**
ICCS	FPGA	23210.95	23129.50	28591.13	44.85	43.30	56.03
	CPU	46499.03	45480.54	49197.40	21.60	21.99	22.98
FBK	CPU	482485.28	502341.93	502798.67	2.08	1.99	2.19
Segmentation							
Node	**AIF**	**E2E Latency (ms)**			**Throughput (fps)**		
		Mean	**Median**	**90th percentile**	**Mean**	**Median**	**90th percentile**
ICCS	FPGA	2168.27	1680.87	4423.50	1.55	0.59	3.12
	CPU	2410.10	2354.44	3944.86	0.55	0.42	1.05
FBK	CPU	4164.45	3387.48	5245.54	0.27	0.30	0.33

Test3:

We deployed both segmentation and classification AIFs (CPU/GPU/FPGA) in the FBK or the ICCS cluster.

Table 17 Integrated CCP Test 3 results

Task	**AIF**	**ICCS MEC System**				**FBK MEC System**			
		Overhead (ms)			**Correctness**	**Overhead (ms)**			**Correctness**
		Mean	**Median**	**90th percentile**		**Mean**	**Median**	**90th percentile**	
segmentation	CPU	1997.7	1789.0	2492.7	yes	1811.6	1823.0	2105.3	yes
	GPU	2548.2	2027.0	4232.8	yes				yes
	FPGA	1861.5	1837.0	2192.6	yes				yes
classification	CPU	1783.7	1767.0	2092.2	yes	1676.4	1634.5	1983.7	yes
	GPU	2014.2	1986.0	2410.6	yes				yes
	FPGA	2643.0	2106.5	4437.2	yes				yes
Sentiment-analysis	CPU				yes	1873.0	1920.0	2223.4	yes

We note that these are application-independent results. The measured overhead is between the different components i.e., MTO, IOC, MEO, IARM, Kubernetes scheduling. It does not include individual AI logic overhead inside the intelligent components in CCP (work that can be included almost independently of the components and can be researched in the future as optimization steps). We also note that the NSAP

leverages the monitoring mechanism present in both MEC systems (representative values shown in Figure 67) so that it gets aggregated values from both (near_edge and far_edge metrics from both MEC systems), in order to decide initial MEC placement of incoming AIF.

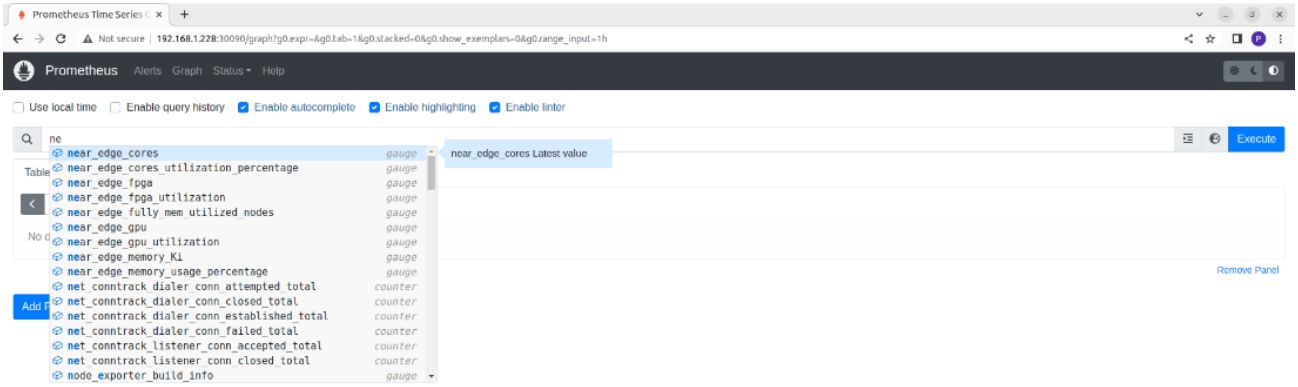


Figure 67 Representative metrics aggregated at the NSAP layer for intelligent MEC selection from IOC

Concluding Remarks on the Validation of Scenario5

Overall, for what concerns the integration of the CCP components, all the AIF deployments tested in scenario 5 were successful. Therefore, primarily, we have demonstrated the correct operation between different NSAP components (MTO and IOC) and CCP components (MEO, IARM, MECMP). Their interfaces operated correctly, messages were exchanged and parsed as planned, and the example AIF was able to be deployed in the end by the underlying Kubernetes at various nodes, accelerated or not, at one or multiple MEC systems. The time overhead to do the deployment over all these CCP components was measured at approx. 1sec when having 1 MEC and approx. 2sec when involving multiple MEC systems. We note that, in the second test, higher overheads were measured both for deployment and AIF performance when merging remote clusters over VPN (e.g., zerotier); however, this effect is attributed to the specifics of the VPN system and not to CCP.

4.6 Scenario6: non-RT RIC

The main objective of this scenario is to validate two main functionalities of the non-RT RIC and its exposure to the rApps: RAN data monitoring and RAN control. Therefore, we considered two different sub-scenarios, as illustrated in Figure 68. The first one was focused on evaluating the Telemetry subsystem of the non-RT RIC, which includes the ICS and the data producers and consumers. The second one comprehended the evaluation of an rApp which dynamically manages the allocation of resources of multi-RAT slices, involving the telemetry and RAN control subsystems.

The non-RT RIC was implemented as a Kubernetes[13] cluster in an Intel NUC9i7QNX with the following specs: 6-Core at 2.6GHz, 64 GB RAM and 1 TB SSD. The OSC components are based on version G.

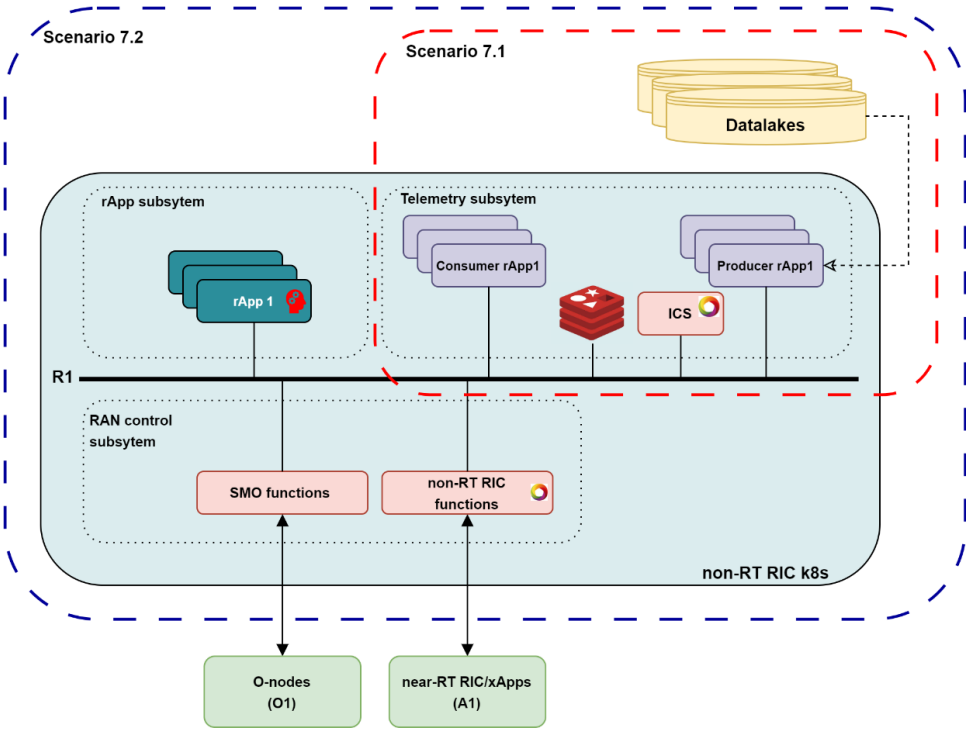


Figure 68 Non-RT RIC evaluation (Scenario 7)

non-RT RIC Telemetry subsystem (sub-scenario 6.1)

As introduced in Section 2.3.4, the ICS decouples data producers and consumers by implementing a data subscription service which manages registered information types and the jobs or subscriptions serving them. Figure 69 illustrates the main workflow involved in this scenario.

[13] https://k3s.io/

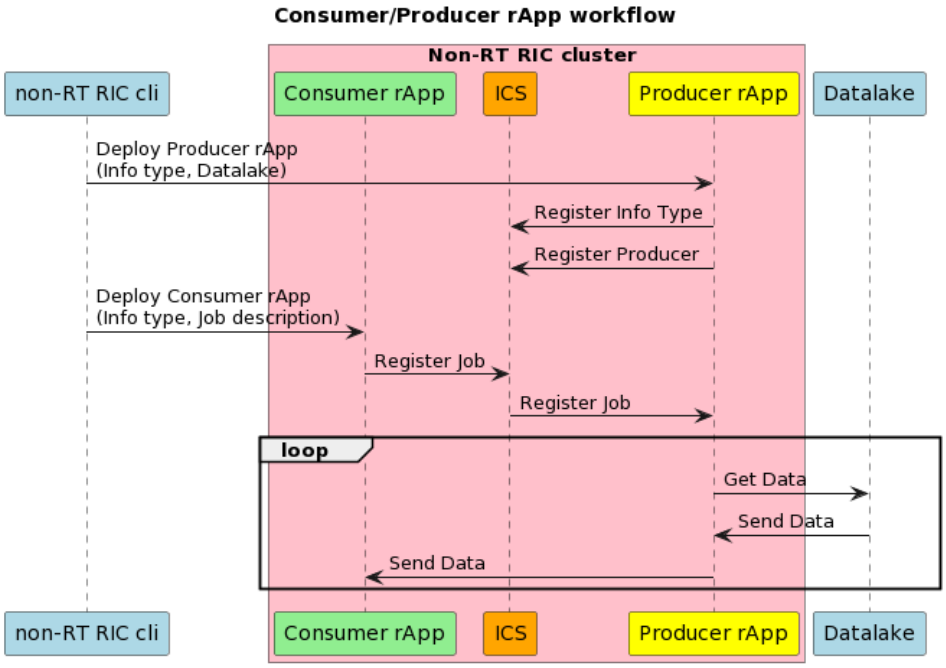


Figure 69 Scenario 7.1 - Main ICS workflow

In our implementation, which is focused on exposing data stored in Prometheus servers, once deployed the producers register in the ICS the information type being served. This information type can contain all or a subset of the metrics stored in a Prometheus server. Then, the deployed data consumers register in the ICS a data subscription or job for each of the required information types. In the description of the job, the consumer specifies the Prometheus metric name, the gathering interval, the labels, or the Prometheus functions to be applied (e.g., sum, average, max…). This way, the producer generates the required query to the Prometheus server and sends the result to the consumer. Figure 70 and Figure 71 exemplify, respectively, the call to deploy a Producer rApp and the associated information type stored in the non-RT RIC and ICS. Figure 72 illustrates the deployment of a Consumer rApp with a job associated with the registered information type.

```
{
   "name":"test-producer",
   "prometheus_url":"prometheus-server-service:9090",
   "info_type":"test-info-1",
   "info_description":"Metrics available: process_cpu_seconds_total, promhttp_metric_handler_requests_total and node_cpu_seconds_total."
}
```

Figure 70 Scenario 7.1 – Producer deployment call

```
{
  "job_data_schema": {
    "$schema": "http://json-schema.org/draft-07/schema#",
    "title": "Prometheus_Metrics",
    "description":"Metrics available: process_cpu_seconds_total, promhttp_metric_handler_requests_total and node_cpu_seconds_total.",
    "type": "object"
  },
  "type_status": "ENABLED",
  "no_of_producers": 1
}
```


Figure 71 Scenario 7.1 – Get Info Type call


```
{
  "name": "test-consumer",
  "info_type": "test-info-1",
  "job_interval": 30,
  "job_metrics": [
    {
      "name": "process_cpu_seconds_total",
      "friendly_name": "sum1d_cpu_seconds",
      "query_function": "sum",
      "query_interval": "1d"
    },
    {
      "name": "node_cpu_seconds_total",
      "friendly_name": "node_cpu_s",
      "labels": {
        "node_id": "node_1",
      }
    }
  ]
}
```


Figure 72 Scenario 7.1 – Consumer deployment call

Evaluation

In order to evaluate the capabilities of the implemented Telemetry subsystem to expose data to rApps with non-RT granularity (i.e., greater than 1 second), we analysed its behaviour under different conditions: (i) multiple consumers demanding the same information type being served by one producer of a single Prometheus Server, (ii) multiple consumers demanding the same information type being served by two/three producers of a single Prometheus Server, (iii) multiple consumers demanding different information types being served by two/three producers of a single Prometheus Server and, (iv) multiple consumers demanding different information types being served by two/three producers of different Prometheus Servers. In all the cases, each deployed consumer created a job with a required period of 1 second and we computed the excess delay as the time difference between the required period and the measured period. New consumers were deployed every 5 minutes.

Figure 73 depicts the excess delay obtained according to the number of producers and the increasing number of consumers. As shown in the figure, with a single producer, the telemetry subsystem suffered congestion with the increasing number of consumers (starting at 25) and the excess delay started to increase linearly. In order to determine if the congestion was caused because of using a single producer, we repeated the scenario with two and three producers. However, we found that in case of having multiple producers of the same information type, the ICS just replicated each consumer job in each of the producers, thus increasing

the congestion in the subsystem (i.e., the excess delay started to increase linearly with a lower number of consumers). Also, note that this behaviour led to some negative excess delays, due to the different producers generating the same data type and sending at different moments to the same consumers. In the case of Prometheus, since new data was only generated each 1 second, it led to the reception of repeated data by the consumers.

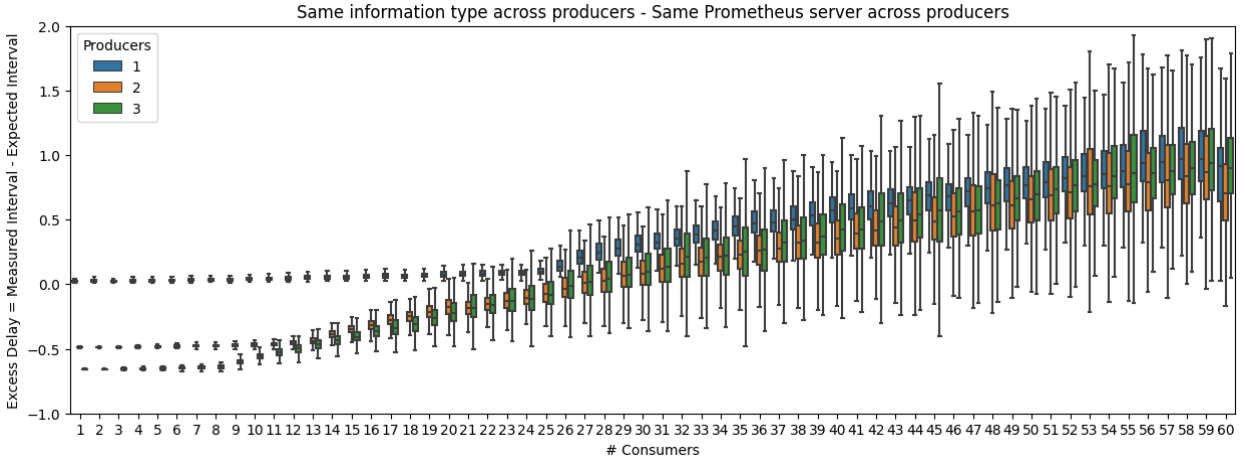


Figure 73 Excess delay - Same information type and same Prometheus server

In the second scenario, we considered the case of different producers serving different information types from a single Prometheus server, in order to avoid receiving replicated data. As shown in Figure 74, this didn't lead to a significant reduction of the excess delay with the number of consumers; indeed, the obtained trend was very similar to the case of one single producer. Thus, we concluded that congestion in the producers was not the main cause of the increasing delays.

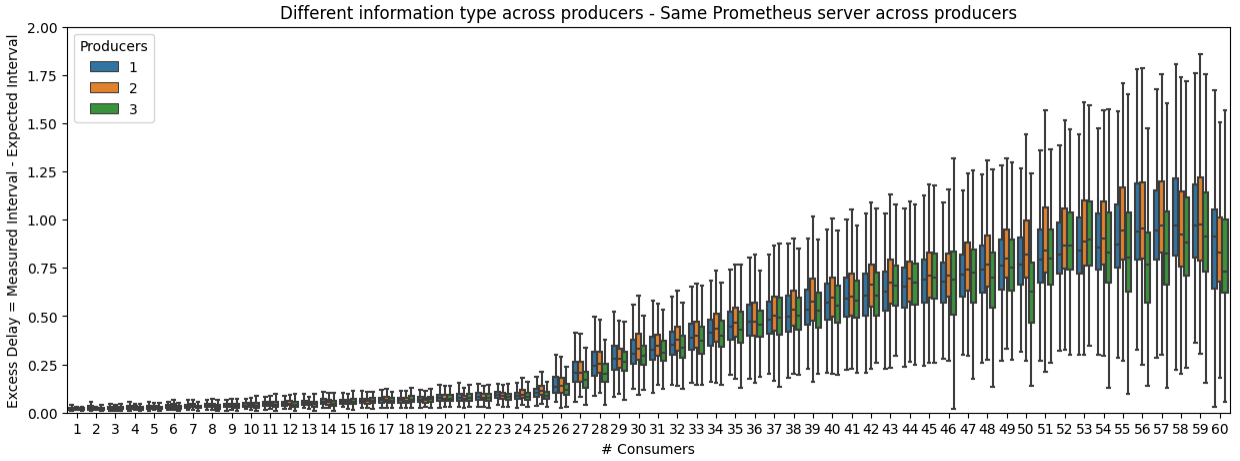


Figure 74 Excess delay – Different information type and same Prometheus server

Finally, we evaluated a scenario where each producer served a different information type and obtained the data from different Prometheus servers. In this case, as depicted in Figure 75, the obtained results indicated

that this solved the congestion and that the multiple HTTP queries to the API of a single Prometheus server were the root cause of the increasing excess delay.

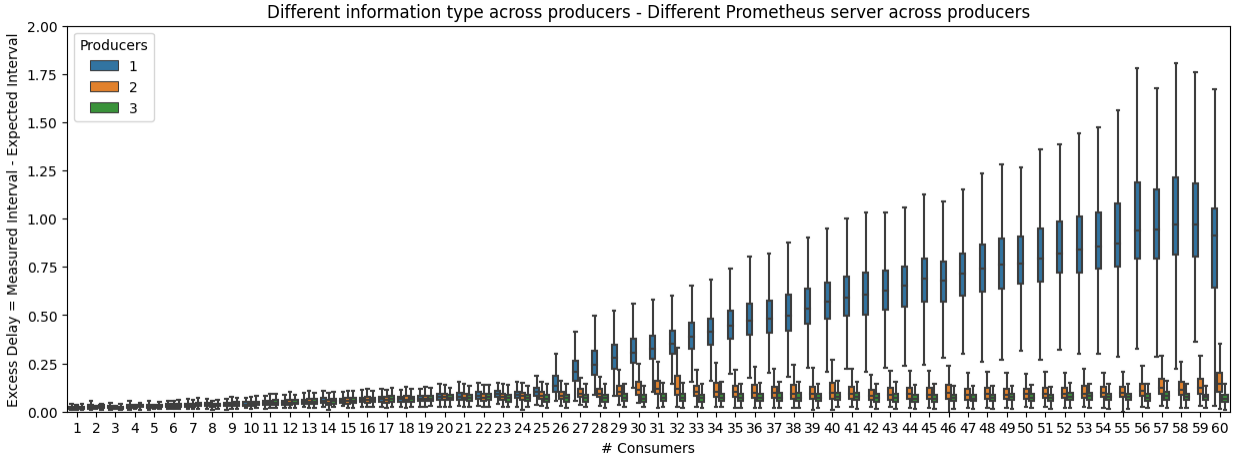


Figure 75 Excess delay – Different information type and different Prometheus server

To conclude, although the evaluated congestion scenario could probably be solved by increasing the computing resources of the Prometheus server, in cases where a specific information type is demanded by several consumers, it could be useful to store the data in a shared database available to the different consumers. This way, data could be accessed in a more efficient way without requiring recurrent queries to a server. As introduced in Section 2.4.2, the implemented non-RT RIC architecture incorporates a REDIS database for this purpose.

Multi-RAT slicing control rApp (sub-scenario 6.2)

This scenario evaluates the functionality of an rApp devoted to the dynamic management of the RAN resources assigned to multi-RAT slices using 5G, 4G and Wi-Fi technologies. The implementation of the rApp is based on the "global scheduler" algorithm presented in our previous work in [33], which, however, was limited to Wi-Fi technology and not integrated within the non-RT framework. Follows a list of the main features of the algorithm; for additional information, the interested reader is referred to [33]:

- A single RAN device (e.g., Wi-Fi AP) can support several slices (e.g., virtual Wi-Fi SSID).
- The RAN resources assigned to the slices in each RAN device are configurable (e.g., airtime in Wi-Fi).
- Slice SLAs define the percentage of resources assigned to an SLA in a geographical area covered by several RAN devices.
- The proposed algorithm (formulated as quadratic programming problem) controls the scheduling weights in individual RAN devices to deliver resource allocations for each slice that fulfil an SLA contract.

- The algorithm makes use of the offered load and consumed resources in the previous period to adjust the weights for the next period.
- The algorithm tries to compensate punctual SLA violations using an Exponentially Weighted Moving Average (EWMA) which defines a time window where the required SLAs should be achieved.

In AI@EDGE, we have extended this algorithm to also integrate cellular RATs, considering the percentage of total PRBs being consumed by the different deployed slices. Then, in the case of multi-RAT slices, the algorithm is able to dynamically balance their resource allocation according to the offered load in the different RATs and in order to fulfil the SLA of the slices over a given geographical area. All RATs are managed by a common SMO, which allows to configure and manage the RAN slices through the O1 interface using NETCONF. The RAN devices implement a Prometheus Exporter, which enable the NSAP to monitor RAN metrics such as the percentage of radio resources, throughput, number of connected users or link quality. These two functionalities, RAN monitoring and RAN control, are exposed to the Slicing rApp by the non-RT RIC as illustrated in Figure 76.

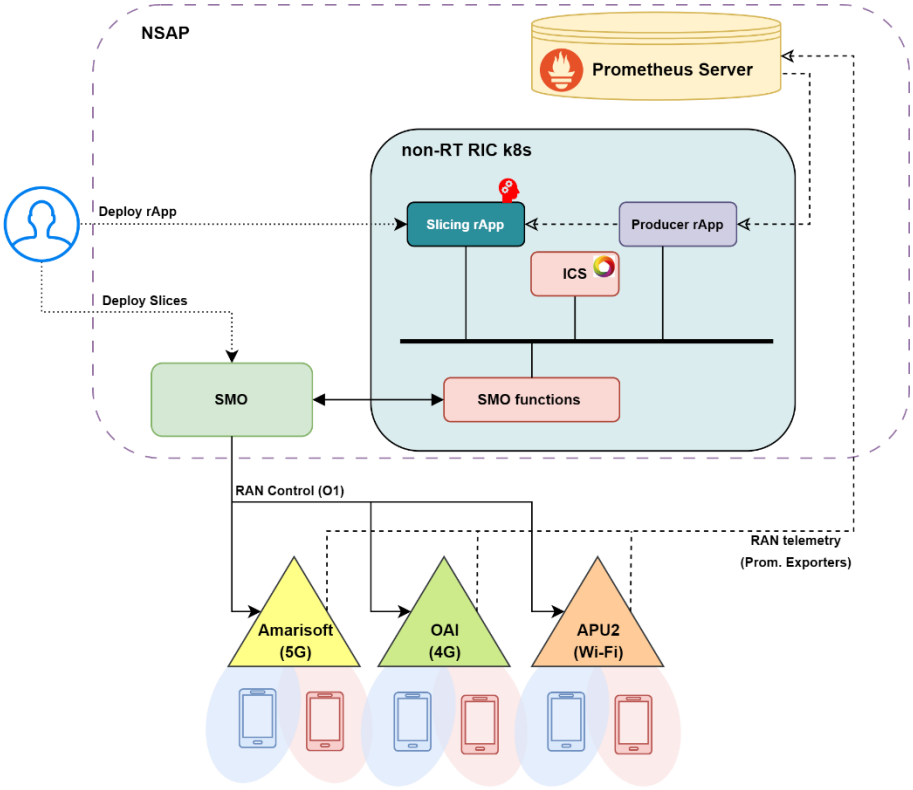


Figure 76 Scenario 7.2 - Involved components

In the evaluation testbed, the 5G connectivity was provided by an Amarisoft Callbox[14], the 4G connectivity by a OpenAirInterface (OAI)[15], and the Wi-Fi connectivity by a Single Board Computer (SBC) APU2 from Pc Engines[16] using hostapd software[17]. Table 18 summarizes the main characteristics and configuration parameters of the RATs.

Table 18 Scenario 7.2 – Testbed RATs description

RAT hardware	**HW details**	**SW details**	**Main RAT configuration**
Amarisoft (5G)	Amarisoft Callbox Advanced + Amarisoft SDRs	Amarisoft FW: 2023-06-10 5G Core: Open5Gs v2.6.4 Custom Prom. Exporter	N77, 4.05 GHz, 50 MHz, 30 KHz, TDD
OAI (4G)	Intel NUC10i7FNH + Ettus B210	OAI RAN: 2022.w42 FlexRAN: v2.4 5G Core: Open5Gs v2.6.4 Custom Prom. Exporter and Local Scheduler	N7, 2.685 GHz, 20 MHz, FDD
APU2 (Wi-Fi)	APU2 apu4d2 + Atheros WLE200NX Wi-Fi	Hostapd v2.7 Custom Local Scheduler Hostapd Prom. Exporter	20 MHz, Channel 149, 802.11n

According to its algorithm, the rApp is able to dynamically configure the resources assigned to the slices (i.e., the % of total resources) in the 4G and Wi-Fi radios through the implementation of local schedulers configurable through external APIs exposed to the SMO. In the case of OAI, the implementation extends FlexRAN slicing functionality[18] to allow us to dynamically manage the PRBs assigned to the slices. In the case of Wi-Fi, the scheduler leverages the work in [34], as presented in [33]. However, Amarisoft doesn't provide a way to externally modify the behaviour of the scheduler, which by default implements proportional fairness among all users independently of the slice they belong to. Therefore, as will be showcased in the evaluation section, in some cases the rApp needed to compensate the resources consumed in the 5G RAT with the 4G and Wi-Fi RATs.

Figure 77 illustrates the main workflow of the Slicing rApp. In this case, the Slicing rApp also works as a Consumer rApp, getting RAN telemetry from the Producer rApp which exposes the data in the Prometheus server. The Slicing rApp periodically computes the resource allocation of the slices according to the RAN telemetry and sends the new allocation to the SMO.

[14] https://www.amarisoft.com/products/test-measurements/amari-lte-callbox/
[15] https://openairinterface.org/oai-5g-ran-project/
[16] https://www.pcengines.ch/apu2.htm
[17] https://w1.fi/hostapd/
[18] https://gitlab.eurecom.fr/mosaic5g/mosaic5g/-/wikis/tutorials/slicing

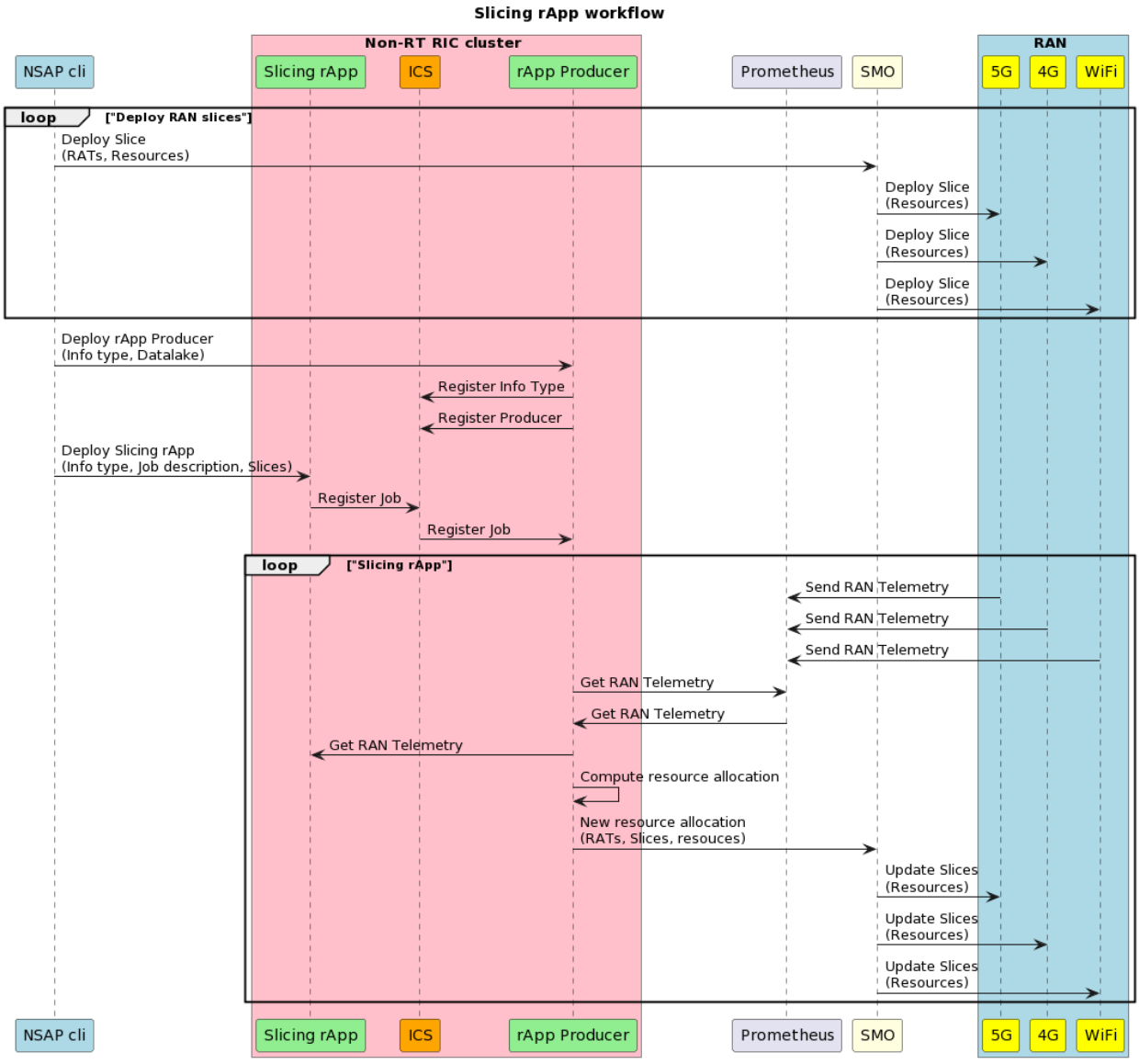


Figure 77 Scenario 7.2 – Main workflow

The objectives and workflows of this solution are aligned with O-RAN's Use Case "RAN Slice SLA Assurance" [35], which aims to ensuring slice SLAs and preventing its possible violations. Note that in a scenario including a near-RT RIC and E2 nodes, the decision of the rApp could be sent as an A1 policy to the xApps managing the local schedulers of the different technologies, as is described in [35].

Evaluation

The evaluation scenarios included one Amarisoft gNB, one OAI eNB and one Wi-Fi AP, as specified in Table 18. We deployed two different slices in each of the RATs and one UE per slice in each RAT, varying the slice SLAs and load required by the UEs (only downlink direction) according to the evaluated scenario.

We implemented three different test cases to analyse the performance of the slicing rApp under different conditions: (i) Fixed load, (ii) dynamic 5G load and (iii) dynamic Wi-Fi load.

In the case of 5G and 4G technologies, we used MOCN to create two different slices (i.e., RAN sharing with two different 4G/5G cores). UE traffic was generated with *iperf* and the sum of the traffic in the two slices saturated each of the RATs (i.e., all the radio resources of the RATs were consumed). The main KPIs such as the consumed resources in each RAT and slice were captured with Grafana by means of the Prometheus Exporters and exported to CSV files to analyse them. The period of the rApp was configured to 10 seconds and the SLA period was 100 seconds (EWMA with alpha 0.1). The minimum resource allocation per slice in a RAN node was configured to 5%.

Figure 78 and Figure 79 show the bodies required to deploy the producer and consumer rApps, respectively. Note that in the case of the consumer rApp, which implements the algorithm to manage the resources assigned to the slices, it is defined the type of information needed; this is used by the ICS to connect it to the producer rApp, as evaluated in the previous section. Also, the referenced instances are the IDs of the deployed slices by the SMO. These IDs are used to obtain from the SMO the required information and endpoints to manage the slices (e.g., type and number of RAN nodes per slice). The airtime weight refers to the SLA (i.e., percentage of RAN resources).


```
{
  "name": "gsa-producer",
  "prometheus_url": "prometheus-server-service:9090",
  "info_type": "gsa-info",
  "info_description": "Metrics available: WiFi airtime by VLAN, 4G airtime by PLMNID and 5G airtime by PLMNID"
}
```

Figure 78 Scenario 7.2 – Producer rApp deployment (RAN telemetry)


```
{
  "name": "gsa-consumer",
  "info_type": "gsa-info",
  "job_interval": 10,
  "instances": [
    {
      "id": "140355037909056",
      "airtime_weight": 30
    },
    {
      "id": "140355064085424",
      "airtime_weight": 70
    },
  ],
  "config": {
    "gsc_endpoint": "192.168.40.114:30000",
    "epsilon": 0.1,
    "lb_constraint": 0.05,
    "eq_sl_weights": 1,
    "chan_eff": 1,
    "bias": 0.01,
    "interval_ms": 10000,
    "solver": "quadprog",
    "debug_level": 3,
    "airtime_threshold": 0,
    "timeout": 3,
    "retries": 3,
    "backoff": 0.3,
    "max_errors": 20
  }
}
```


Figure 79 Scenario 7.2 – Consumer rApp deployment (RAN slicing)

1. Fixed load scenario:

The first scenario was focused on the functional validation of the rApp in cases without load variations. We considered slice SLAs demanding different resource allocations and analysed the decisions of the rApp and their impact on the RAN. Also, we compared results with a static case, where the resource allocation was locally fixed per RAN and slice during the slice deployment. Each test lasted for 2-3 minutes.

Figure 80 shows the variation of the consumed resources in the different technologies (5G, 4G and Wi-Fi) and slices as reported by their different RAN exporters, for a static scenario targeting SLAs of 75% (Slice 1) and 25% (Slice 2) of the global resources. Also, it depicts the global resource allocation of each slice computed as the average of the three technologies. In this case, as aforementioned, the 5G resource allocation using Amarisoft Callbox hardware could not be dynamically modified and remained at 50% for each slice. Therefore, the rApp compensated it with 4G and Wi-Fi RANs to meet the required SLAs. Thus, it outperforms the static allocation.

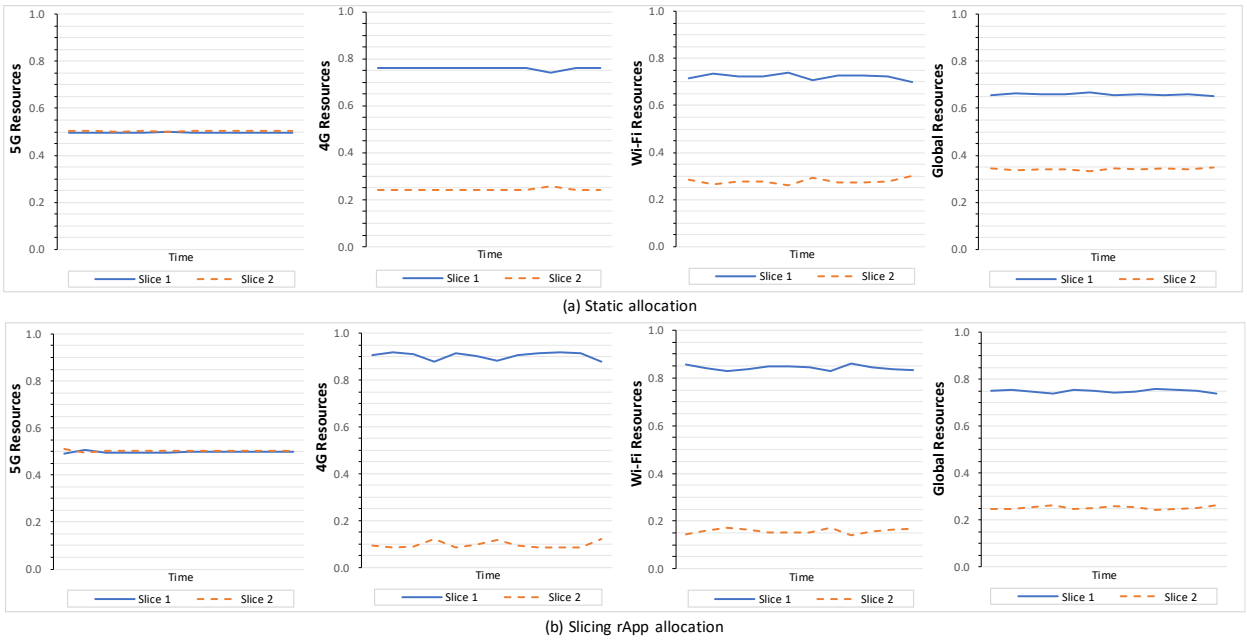


Figure 80 Scenario 7.2: SLA 75%-25%, fixed load: (a) Static allocation and (b) Slicing rApp allocation.

Table 19 and Table 20 summarize the results of the different static scenarios, showing the average resource allocations performed during the scenario duration and according, respectively, to the static allocation and the slicing rApp. As aforementioned, the slicing rApp was able to dynamically modify the allocation of Wi-Fi and 4G slices to compensate for the 5G fixed allocation, achieving in almost all the cases the required SLAs; only in the last scenario, which comprehended an SLA of 80%-20%, the Wi-Fi RAT was not able to achieve the required resource allocation of approximately 95%-5% due to limitations in its local scheduler implementation. In any case, the rApp outperformed the static allocation in all the cases, avoiding SLA errors up to the 10%.

Table 19 Scenario 7.2: All SLAs, fixed load: Static allocation.

SLA	5G Slice 1	5G Slice 2	4G Slice 1	4G Slice 2	Wi-Fi Slice 1	Wi-Fi Slice 2	Global Slice 1	Global Slice 2	SLA Error
50%-50%	48.1%	51.9%	50.0%	50.0%	50.1%	49.9%	**49.4%**	**50.6%**	**0.6%**
55%-45%	49.6%	50.4%	56.0%	44.0%	54.3%	45.7%	**53.3%**	**46.7%**	**1.7%**
60%40%	49.6%	50.4%	60.0%	40.0%	59.1%	40.9%	**56.2%**	**43.8%**	**3.8%**
65%-35%	49.5%	50.3%	66.7%	33.3%	64.3%	35.7%	**60.2%**	**39.8%**	**4.8%**
70%-30%	49.6%	50.4%	69.7%	30.3%	67.7%	32.4%	**62.3%**	**37.7%**	**7.7%**
75%-25%	49.6%	50.4%	75.8%	24.2%	72.2%	27.8%	**65.9%**	**34.1%**	**9.1%**
80%-20%	49.7%	50.5%	80.3%	19.7%	78.5%	21.5%	**69.5%**	**30.5%**	**10.5%**

Table 20 Scenario 7.2: All SLAs, fixed load: Slicing rApp allocation.

SLA	5G Slice 1	5G Slice 2	4G Slice 1	4G Slice 2	Wi-Fi Slice 1	Wi-Fi Slice 2	Global Slice 1	Global Slice 2	SLA Error
50%-50%	49.7%	50.3%	50.6%	49.5%	49.7%	50.3%	**50.0%**	**50.0%**	**0.0%**
55%-45%	49.6%	50.4%	57.6%	42.4%	56.7%	43.3%	**54.7%**	**45.3%**	**0.3%**
60%40%	49.7%	50.3%	66.7%	33.3%	64.1%	36.0%	**60.1%**	**39.9%**	**0.1%**
65%-35%	49.7%	50.3%	73.8%	26.3%	71.1%	28.9%	**64.8%**	**35.2%**	**0.2%**
70%-30%	49.7%	50.4%	82.4%	17.6%	78.0%	22.0%	**70.0%**	**30.0%**	**0.0%**
75%-25%	49.7%	50.3%	90.5%	9.5%	84.4%	15.6%	**74.9%**	**25.1%**	**0.1%**
80%-20%	49.7%	50.4%	95.9%	4.2%	88.9%	11.1%	**78.1%**	**21.9%**	**1.9%**

2. Dynamic 5G load:

In the second test case, we evaluated a scenario where the 5G load of the different slices was modified, leading to different resource allocations in the 5G RAT: 50%-50%, 60%-40%, 80%-20%, 40%-60% and 20%-80%. Each allocation period lasted for 2-3 minutes. This is depicted in the "5G Resources" graph of Figure 81. The Global SLA error or violation is computed according to the obtained SLAs applying the defined EWMA and comparing it to the required SLA, as follows:

$$SLA_{error}(t) = |SLA_{req} - SLA(t)|,$$
$$where\ SLA(t) = 0.1 \times SLA_{meas} + 0.9 \times SLA(t-1)$$

In the case of the static allocation, as depicted in Figure 81a, since the 4G and Wi-Fi resource allocation remained static during time, the global resource allocation was not able to meet the SLAs of the slices; indeed, in the final allocation period of the experiment, Slice 2 obtained more resources than Slice 1, leading to an SLA error greater than 10% (in each slice). On the other hand, as illustrated in Figure 81b, the slicing rApp was able to compensate for 5G load variations by dynamically modifying the resources allocated to the 4G and Wi-Fi, leading to SLA violations lower than 2%. Note that since the rApp reacts to the measured resource allocation after each algorithm running period (10s in this experiment), performed optimizations cannot avoid some punctual SLA errors; however, lower rApp periods could be applied in case a faster optimization is needed.

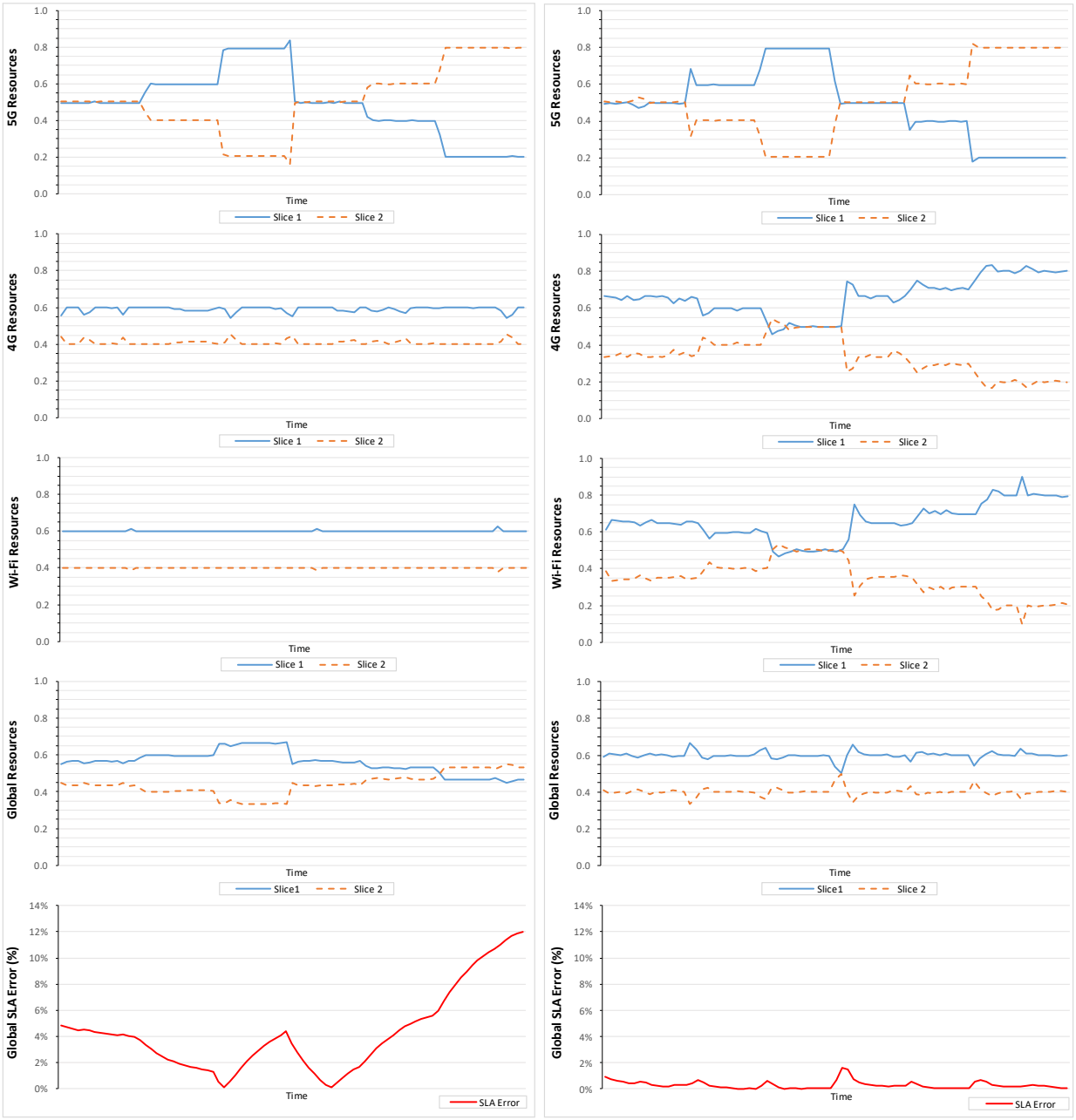


Figure 81 Scenario 7.2: SLA 60%-40%, dynamic 5G load: (a) Static allocation and (b) Slicing rApp allocation.

3. Dynamic Wi-Fi load:

Finally, the third test evaluated a corner case where, due to load variations (i.e., traffic was stopped in the first slice of the Wi-Fi AP for a period), the rApp was not able to fulfil the SLA by modifying the resource allocation in the 4G slices. As shown in Figure 82b, during this period, the SLA error of the rApp increased up to the 10%. Then, after starting again the load of Slice 1 in the Wi-Fi RAT, the rApp tried to compensate

the SLA violation by giving additional resources to Slice 1 in the 4G and Wi-Fi RATs, leading to a distribution of global resources of approximately 80%-20%. This wanted behaviour, which is caused by the definition of the SLA using the EWMA, led to a fast reduction of the SLA error. Finally, the resource allocation in all the slices stabilized to the prior values before the violation. In the case of the static allocation, as shown in Figure 82a, the SLA error increased up to 21% during the period without the Wi-Fi traffic in Slice 1, and it decreased slowly after it became again activated.

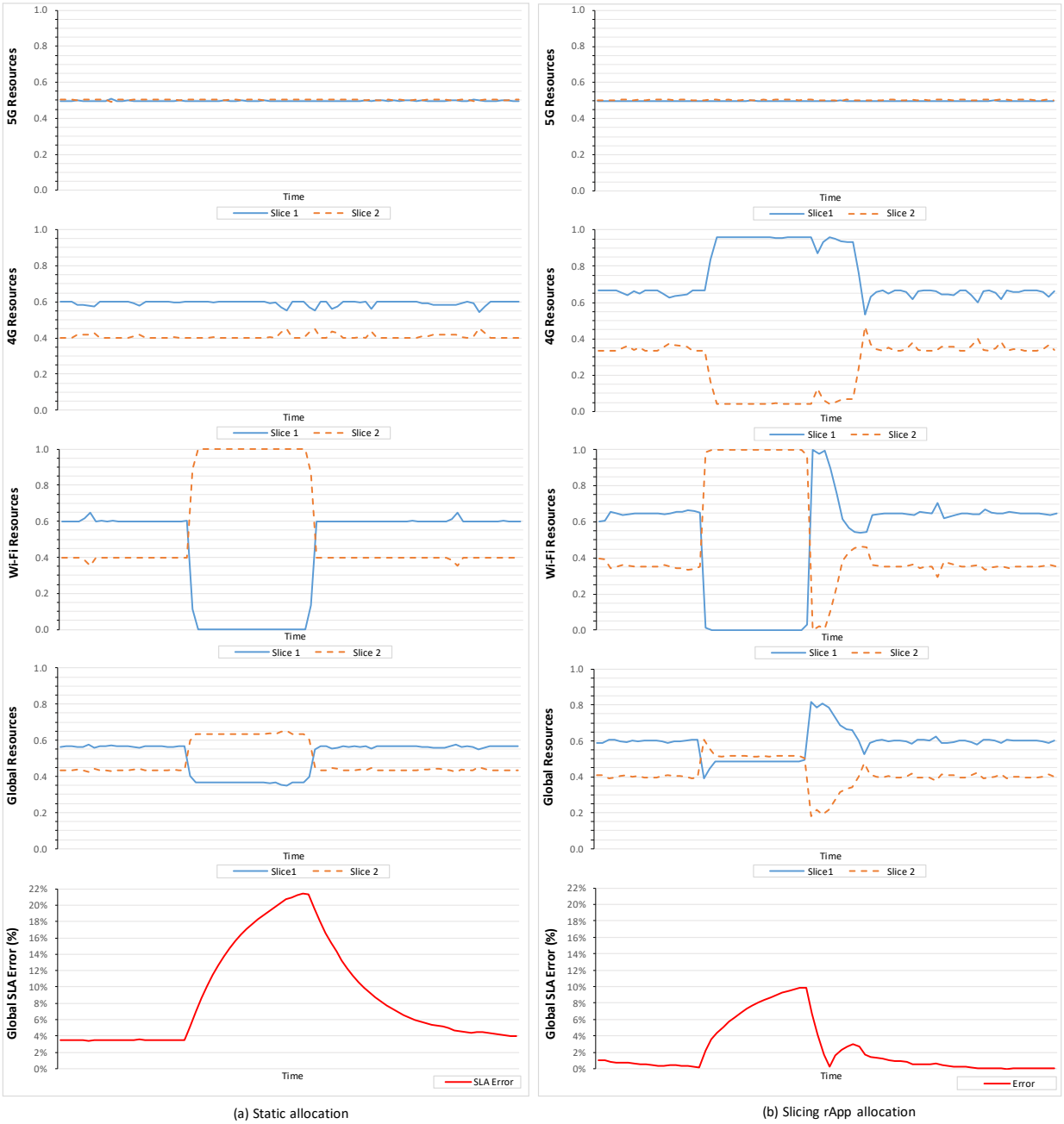


Figure 82 Scenario 7.2: SLA 60%-40%, dynamic Wi-Fi load: (a) Static allocation and (b) Slicing rApp allocation.

Concluding Remarks on the Validation of Scenario6

Overall, in the validation of non-RT RIC, the above results validate the implementation of rApps using the non-RT RIC Telemetry and RAN control subsystems, enabling the operation of non-RT intelligent control-loops. First, we evaluated the capacity of O-RANs's ICS to expose non-RT data to rApps using decoupled data consumers and producers. Results were satisfactory, being the implemented telemetry subsystem able

to cope with an increasing number of data consumers and without impacting the execution of the non-RT control-loops. Secondly, we developed an rApp which aim to dynamically managing and optimizing the radio resource allocation of multi-RAT slices with a non-RT granularity and according to defined SLAs. We evaluated the rApp in an experimental testbed involving AI@EDGE's non-RT RIC and multiple slices and RATs (5G, 4G and Wi-Fi). Results showed that the rApp was able to meet the SLAs by dynamically adapting the radio resources of the different RATs according to traffic variations in the deployed slices, outperforming static radio resource allocations.

4.7 Scenario7: LCM: Model Management and Model Update

By following the methodology described in Section 3.3.8, we propose the results of the validation process we produced for the Model Management and Update scenario. Here follows a description of the steps we took to orchestrate the process, the measurement and metrics we collected with an analysis of the results and a 'lesson-learned' section regarding the possible application scenario in the AI@EDGE perspective.

Procedure:

- Testbed preparation: We deployed all the necessary components to deploy an AIF function. We created a dedicated namespace for the deployments, deployed the CCP with Prometheus, LightEdge, and the Stand-alone version of the MEO, deployed SeldonCore, an MLFlow Server, and a MinIO Storage.
- Saving the Model: We saved a generic machine learning model built with Scikit Learn in MLFlow format on a MinIO storage system using a Python script. This ensured proper version control and easy access to the model for deployment. This is also a requirement for the Seldon Deployment which fetches the ML model from remote storage during the initialization of the deployment.
- Creating Deployment Plans with Helm Charts: We generated two different deployment plans using Helm charts. One plan was for deploying a simple Seldon deployment, in particular we reused the one proposed in the 3.3.4 validation scenario. The second, for the model auto updater deployment. The model auto updater is described in more detail in Section 2.4.3 of this document. We uploaded the Helm Charts on the helm-chart-repository hosted at GitLab.
- AIF Descriptor: We created an Artificial Intelligent Function (AIF) descriptor for both deployment solutions. The descriptor contained essential information about the settings required for the AIF services. The AIF descriptors were stored on the helm-chart-registry AIFs catalogue and made available for the MEO.
- Deployment with MEO: We deployed the AIF services using a standalone MEO version on the FBK testbed. The deployment took place in the dedicated namespace.
- Measuring Deployment Time: We measured the time it took for the deployment to be ready by using a Python script to check the status of the deployed pods. This allowed us to understand how long it took for the AI function to be fully available for use. We measured the time delta between the scheduling and the readiness of the pods.

- Update model: We relaunched the procedure of the second step to have an updated instance of the same model. MLFlow registry took care of the model versioning.
- Measuring redeployment time: When the ML model is updated, we redeployed the simple Seldon deployment. The redeployment was performed in a semi-automated way through a Python script.

Metrics and analysis

Deployment time

We performed the measurements of deployment time through a Python script that connects to the Kubernetes APIs and retrieves the deployment times for each pod. The measurement is based on the metrics of the pods' PodScheduled and Ready states. We executed the script for both deployments, obtaining the following results:

Table 21 Building time measurement for Model-update-sidecar and Seldon-build

	Model-update-sidecar (sec)	Seldon-build (sec)
	25.00	26.00
	25.00	25.00
	25.00	26.00
	26.00	25.00
	25.00	26.00
	26.00	25.00
	25.00	25.00
	25.00	26.00
	26.00	25.00
	26.00	26.00
Mean time	**25.40**	**25.50**

As we see, there is no substantial difference between the deployment time of the two different use cases. Both the model updater and the Seldon build fetch the ML model from the remote storage at the beginning of the deployment. For the auto updater we used a personalized InitContainer that we reused as monitor sidecar.

Redeployment time

The second step is to measure the redeployment time. We had redeployment time only in the case of the seldon-build deployment. The time took for a plain deployment is the same (~ 25 seconds). To this, we need to add the time needed to send the signal to the MEO for tearing down the previous AIF and the signal to instantiate the new deployment. This timing depends basically on the speed of the connections.

In the case of the model auto updater, we do not have redeployment time because there is no redeployment. The sidecar container in the Seldon pod monitors the MLFlow registry every N second to determine if there

are new model versions available. If it is the case, it downloads in the background the new model and calls an internal MLServer API to dismiss the old version and reload the new one.

In synthesis, the measurements are these:

Table 22 Redeployment times in seconds

Deployment	Time (sec)
Seldon-build	~25 + Variable amount of time depending on connection speed
Model-update-sidecar	0

Disk Space Usage

The metrics for the containers are the same as shown in Section 4.4. The only difference is that we need to add the sidecar container for the model auto updater.

Table 23 Image sizes in MB

Image	Size (MB)
registry.gitlab.com/aiatedge/helm-chart-registry/sidecar-model-update:v1	1140

Concluding Remarks on the Validation of Scenario7

Overall, as observed from the results obtained by measuring various aspects of the deployment lifecycle, there is a substantial time saving by using the Auto Updater model solution. Typical application scenarios for the auto update monitoring solution are, for example, scenarios where new model versions are constantly produced or, more in general and aligned with the spirit of zero-touch automation of AI@EDGE, the case where the administrator of the CCP does not have to intervene manually in the AIF redeployment. The presented approach relies on the implementation of typical MLOps process, where the model is explicitly registered, versioned, and referenced for the deployment. In the scenarios, where AIFs are used for serving the ML inference models, this approach may be fully automated and integrated in CCP implementation, including the model registry and the sidecar deployment together with the model server.

4.8 Scenario8: LCM: Auto-configuration

In this section, we will present the results of the scenario described in section 3.3.9. The aim is to demonstrate the relevance of meta-learning in the reconfiguration of a NIDS by evaluating it based on two criteria: the performance of the NIDS in distinguishing an attack flow from a benign flow, and the speed at which meta-learning can determine the configuration. For evaluation purposes, a positive sample corresponds to an attack flow and a negative sample to a normal flow. We calculate four evaluation metrics: MCC, f1-score, precision, and recall.

Detection Performance:

Ten out of the seventeen days are excluded from the report because the detection performance for these days—across all configuration techniques—exceeds 0.99. Thus, on these days, meta-learning performs on par with both BO and the default setting. On day 12, despite different inferred configurations (in terms of hyperparameter values), the three methods yield identical performance levels, as indicated by metrics such as MCC (0.801), f1-score (0.783), precision (1.0), and recall (0.644). This implies that the NIDS generated classify both attack and regular traffic flows similarly. Comparing to the default settings, on days 2, 3, 8, 9, and 14, BO enhances the MCC by an average of 27.79% (ranging from 5.54% on day 2 to 80.24% on day 9). It's noteworthy that this improvement aligns with BO's goal to optimize this particular metric. Meanwhile, meta-learning also boosts the MCC substantially with an average growth of 25.28% (ranging from 4.47% on day 2 to 79.01% on day 9). Even though precision wasn't a metric of concern in BO's objective function, it saw an average rise of 62.03% (peaking at a 220% improvement on day 9). For meta-learning, this metric improved by 58.69%. As for the recall metric, both BO and meta-learning registered gains on day 14, with BO's recall increasing from 0.767 to 0.97 and meta-learning's from 0.767 to 0.946. Day 13 observed meta-learning's least efficient performance. In contrast, BO enhanced the performance of the NIDS across three out of the four metrics: MCC's default configuration rose from 0.771 to 0.847, f1-score from 0.769 to 0.855, and recall from 0.714 to 0.865. While BO increased precision slightly from 0.833 to 0.847, meta-learning excelled with a precision score of 1.0. However, this came at the expense of a diminished recall score, indicating that while the system did not raise any false alarms, it missed numerous attacks. Day 13 is unique in having a low flow count combined with a significant imbalance between regular and attack traffic, which might heavily influence the constructed meta-features, impacting the learning phase of the meta-model.

Resource usage:

We exclude the time required for learning the meta-model since it's precomputed. In our setup, a new dataset emerges daily. After employing BO 16 times, we aim for a swift configuration on the 17th day. Meta-learning's configuration inference time matches that of a meta-model (our Random Forest regressor). Unlike BO, meta-learning first extracts meta-features from the new dataset. We gauge meta-learning and BO configuration generation time by the count of BO iterations. For meta-learning, we equate the time spent on meta-feature extraction to one BO iteration. Although the precise BO iteration count to reach peak NIDS performance is unpredictable, we use iterations needed for BO to match meta-learning's detection rate.

Figure 83 illustrates that meta-learning consistently operates between 5 and 8 iterations. Even on days where BO marginally outperforms (3-5 iterations), the practical NIDS impact is minimal, meaning meta-learning might take at most 30-50 seconds longer. Conversely, BO frequently demands more iterations, sometimes exceeding 100, making meta-learning on average 9 times faster. Predicting BO iterations beforehand is challenging.

In our method, post-configuration, the model undergoes training on a test set to determine flow maliciousness. BO integrates this with the optimization process, but our meta-learning-based NIDS also considers model training and prediction times. Training is consistent, taking 3-5 iterations, while prediction

takes 1-2. In typical scenarios where BO iteration count isn't pre-determined, BO takes 300 iterations to build a NIDS and predict new flows, whereas meta-learning takes 9-15 iterations.

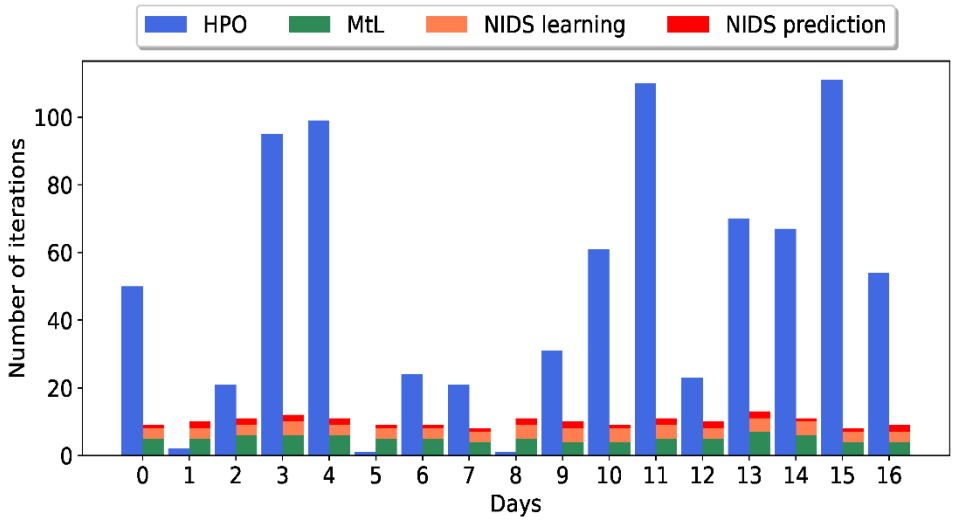


Figure 83 Configuration generation, learning and prediction time of the NIDS

Concluding Remarks on the Validation of Scenario8

Overall, in what concerns auto-configuration, to avoid performance degradation of an AIF, adapting its configuration to a new context is necessary. Using common HPO techniques to reconfigure the AIF is costly. In this scenario, we proposed an alternative that leverages meta-learning to infer a configuration automatically from past experiences. Costly optimizations were done offline to model these past experiences and a new configuration can be then inferred instantaneously. Our technique can be considered as an approximation, but the results provided demonstrate an acceptable degradation of detection accuracy limited to a few percents while the configuration is 9 times faster.

4.9 Scenario9: Model Monitoring

Based on the methodology described in Section 3.3.9 the results obtained in the validation process are presented here for the Model Monitoring scenario. First, a brief description of the components deployed, and the data used are described. Secondly, some of the metrics obtained are presented and analysed to support the validation procedure. Finally, the conclusion and lessons learned are depicted in the last part of the section.

Components:

- Reused components: Some of them are carried over from the previous subsection 4.8, in this scenario will be utilized for storing the model and the detector and monitoring its versions. These components are the instances of MinIO and MLFlow located at the NSAP. All the presented components are deployed at the FBK testbed. It is also important to note that a specific namespace

called “drift” has been created in both clusters (NSAP and MEC) for the deployment of these components.

- New components: We are introducing two new instances within the NSAP, one of MySQL and another of Grafana. For completing the validation process the MMAIF is deployed through MTO & IOC at the MEC system. Furthermore, the training AIFs are also deployed to have the corresponding model and detector stored at the MLFlow/MinIO that will be served by the MLServer of the MMAIF to facilitate inference operations.

Data:

The MySQL database serves as a critical repository for four distinct categories of data:

- Training data: The training data is utilized for both model training and initializing the detector. It comprises two datasets:
    - Original Images Dataset: This dataset contains a collection of unaltered images.
    - Corrupted Images Dataset: Corresponding to the same set of images but intentionally affected by corruption.
- Both datasets are stored within a table named “CIFAR10TrainData” in separate columns:
    - “Xoriginal” for the original images.
    - “XGaussNoise” for the corrupted images.
- Labels associated with these datasets are also stored within the “CIFAR10TrainData” table in a column titled “GroundTruth.” Each dataset comprises a total of 50,000 samples, with each sample stored as a separate row in Binary Large Object (BLOB) type.
- Test/Inference data: This data is instrumental during the testing phase of the model training process and for making inferences using the trained model and detector. Similar to the training data, it includes both original and corrupted image datasets, mirroring the structure of the training data. These datasets are housed within a distinct table called “CIFAR10TestData,” and each dataset consists of 10,000 samples. The “Features” column within the “CIFAR10TestData” table accommodates both original and corrupted image datasets, with the “GroundTruth” column containing corresponding labels. The initial 10,000 images are the original versions, followed by another 10,000 that have been corrupted. Consequently, the “GroundTruth” values are identical for both sets of images.
- Model output data: This category comprises data representing the predicted values generated by the model during inference. Predicted values are stored within the “CIFAR10TestData” table in the “PredictedValue” column.
- Accuracy data: Accuracy data is derived from a comparison between predicted values and ground truth labels, with data used for inference being appropriately labelled, as described in Section 3.3.9. Upon each prediction made using the test data, the “Accuracy” column within the “CIFAR10TestData” table is updated. If the predicted value matches the corresponding

"GroundTruth" value, the "Accuracy" value is set to 1; otherwise, it is set to 0. Subsequently, accuracy is computed by dividing the count of rows where the "Accuracy" column equals 1 by the total count of non-null values in this column. Finally, the resulting accuracy score is stored in a separate table named "CIFAR10Validation" within a column also named "Accuracy."

Procedure:

- A Kubernetes pod called "create-fill-table" has been deployed at the NSAP for the purpose of:
  - Creating the three aforenamed tables: CIFAR10TrainData, CIFAR10TestData and CIFAR10Validation.
  - Download the CIFAR10 dataset.
  - Create the corrupted dataset applying Gaussian Noise to these images.
  - Store the datasets in the CIFAR10TrainData and CIFAR10TestData.

```
aiatedge@nsap:~$ kubectl logs -n drift create-fill-table --follow
2023-09-22 07:55:23.058265: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine
2023-09-22 07:55:23.139328: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine
2023-09-22 07:55:23.139666: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optim
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate comp
2023-09-22 07:55:24.674573: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not fi
2023-09-22 07:55:28,458 - INFO - Downloading original data and creating the corrupted...
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 7s 0us/step
2023-09-22 07:56:09,467 - INFO - Data downloaded and created.
2023-09-22 07:56:09,467 - INFO - Connecting to DataBase...
2023-09-22 07:56:09,475 - INFO - Connection established.
2023-09-22 07:56:09,475 - INFO - Creating MM_VALIDATION.CIFAR10TrainData table...
2023-09-22 07:56:09,508 - INFO - Creating MM_VALIDATION.CIFAR10TestData table...
2023-09-22 07:56:09,537 - INFO - Creating MM_VALIDATION.CIFAR10Validation table...
2023-09-22 07:56:09,560 - INFO - Tables created.
2023-09-22 07:56:09,560 - INFO - Starting to fill tables.
2023-09-22 07:56:09,560 - INFO - Filling training table...
2023-09-22 07:56:11,347 - INFO - Set 1 of 1000 rows inserted.
2023-09-22 07:56:12,307 - INFO - Set 2 of 1000 rows inserted.
2023-09-22 07:56:13,278 - INFO - Set 3 of 1000 rows inserted.
2023-09-22 07:56:15,492 - INFO - Set 4 of 1000 rows inserted.
2023-09-22 07:56:17,008 - INFO - Set 5 of 1000 rows inserted.
```

Figure 84 Log of the pod that creates and populates the tables

```
aiatedge@nsap:~$ kubectl get pods -n drift
NAME                     READY   STATUS      RESTARTS   AGE
create-fill-table        0/1     Completed   0          22m
grafana-6b784d8bb8-qxbp5 1/1     Running     0          56m
mysql-0                  1/1     Running     0          57m
aiatedge@nsap:~$
```

Figure 85 Pod properly executed

- After the pod execution, both training AIFs for the model and the detector are triggered through MTO to train and initialize them respectively and push the resulting model and detector to MLFlow instance at the NSAP. It's important to note that four different training charts and its corresponding descriptors have been created in order to train the detector and the model with unaltered and corrupted data. The initial two will be executed presently, while the remaining two will be triggered upon drift detection. In an ideal scenario, newly acquired data should be continuously stored in the

MySQL database. The training functions could then access the most recent data to facilitate the training process, obviating the need for multiple instances of the AIF for training. However, for validation purposes, we have adopted the current approach. The descriptors are named as "AIF_v4_classifier_training_aif_ori_descriptor.yml" and "AIF_v4_mmd_training_aif_ori_descriptor.yml" for retrieving the original data, and "AIF_v4_classifier_training_aif_cor_descriptor.yml" and "AIF_v4_mmd_training_aif_cor_descriptor.yml" for corrupted data.

```
aiatedge@nsap:~$ curl -X 'POST' 'http://192.168.49.1:30543/meo/aifd/deploy/' -H 'accep
t: application/json' -H 'Content-Type: application/json' -d '{"aifd_name": "AIF_v4_mmd
_training_aif_ori_descriptor.yml", "namespace": "drift"}'
{"aifd_id":"65116e8b772e5c460e53d7e3","bodytosend":{"release_name":"mmd-training-aif-o
ri-65116e8b772e5c460e53d7e3","repochart_name":"aiatedge/mmd-training-aif-ori","values"
:{"nodeSelector":{"kubernetes.io/hostname":"cluster-master"}}},"command":{"namespace":
"drift"},"id":"65116e8c740286a1e1f18ec9","name":"mmd-training-aif-ori-65116e8b772e5c46
0e53d7e3"}
aiatedge@nsap:~$ curl -X 'POST' 'http://192.168.49.1:30543/meo/aifd/deploy/' -H 'accep
t: application/json' -H 'Content-Type: application/json' -d '{"aifd_name": "AIF_v4_cla
ssifier_training_aif_ori_descriptor.yml", "namespace": "drift"}'
{"aifd_id":"65116efc772e5c460e53d7e6","bodytosend":{"release_name":"classifier-trainin
g-aif-ori-65116efc772e5c460e53d7e6","repochart_name":"aiatedge/classifier-training-aif
-ori","values":{"nodeSelector":{"kubernetes.io/hostname":"cluster-worker1"}}},"command
":{"namespace":"drift"},"id":"65116efd740286a1e1f18eca","name":"classifier-training-ai
f-ori-65116efc772e5c460e53d7e6"}
```

Figure 86 Training AIFs deployment through MTO

- After the successful completion of the previous step, the initialization process is marked as finished. Subsequently, in accordance with the description provided in Section 3.3.9, two distinct procedures are executed to assess the performance of a monitored model in comparison to an unmonitored one. In the unmonitored procedure, no additional actions are taken beyond inference. In this discussion, we will focus uniquely on the scenario where the model is being actively monitored. Thus, this marks the point at which the MMAIF can be deployed.

```
aiatedge@nsap:~$ curl -X 'POST' 'http://192.168.49.1:30543/meo/aifd/deploy/' -H 'accep
t: application/json' -H 'Content-Type: application/json' -d '{aifd_name: AIF_v4_sid_up
d_monit_aif_descriptor.yml, namespace: drift}'
{aifd_id:6504713562b0a311141a10c3,bodytosend:{release_name:sid-upd-monit-aif-650471356
2b0a311141a10c3,repochart_name:aiatedge/sid-upd-monit-aif,values:{nodeSelector:{kubern
etes.io/hostname:}}},command:{namespace:drift},id:65047137e27ab1b12047713e,name:sid-up
d-monit-aif-6504713562b0a311141a10c3}
                    $ kubectl get pods -n drift
NAME                                                READY   STATUS    RESTARTS   AGE
sid-upd-monit-aif-inference-system-dep-5d594d5d5f-7s6cq   1/1     Running   0          8d
sid-upd-monit-aif-mlserver-system-dep-f7cb9489f-tbv4n     2/2     Running   0          8d
```

Figure 87 MMAIF deployment through MTO

- With both the model and detector now being served by the MLServer instance, a Python script has been developed to facilitate inference operations. The script is designed to make inferences on both the model and detector, employing different batch sizes for each; 10 for the model and 1000 for the detector. To optimize the inference process, a buffering mechanism has been implemented. When the script calls the model for inference, it stores input data in a buffer. This buffer accumulates data

until it reaches a capacity of 1000 values. Only when this buffer is full, the script proceeds to perform inference on the detector using the buffered data. This approach helps efficiently manage the inference process, ensuring that the detector operates on a batch of 1000 values for each inference cycle.

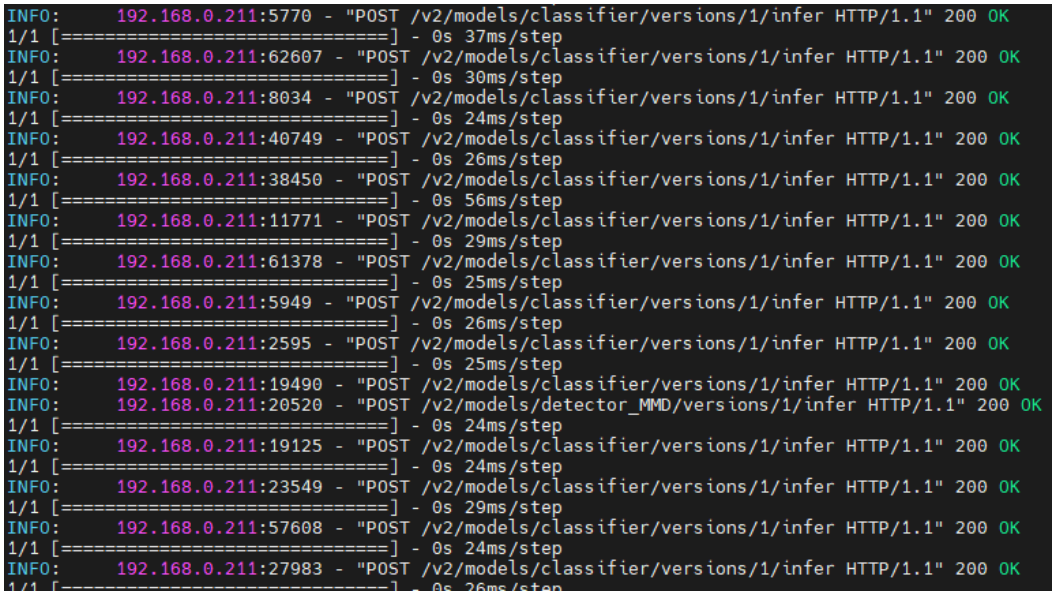


Figure 88 MLServer pod logs showing inference done over the classifier and detector

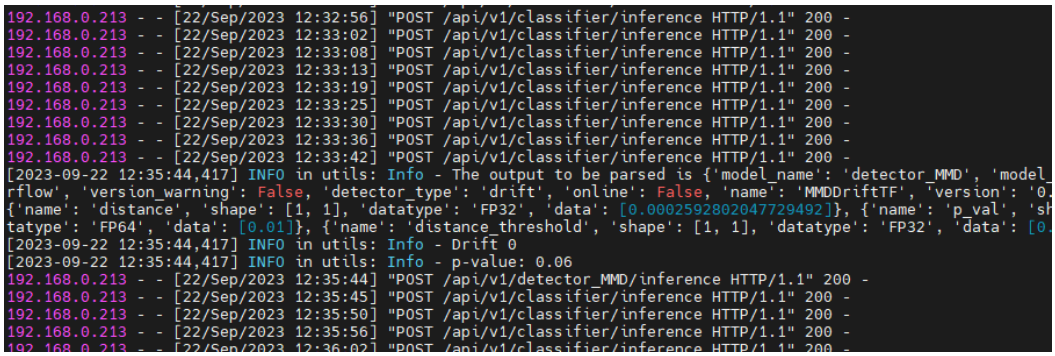


Figure 89 Inference system pod logs showing the flask app calls for inference over the classifier and detector

- The inference script retrieves data from the MySQL database by querying the “Features” column of the “CIFAR10TestData” table. This data serves as input for making inferences using both the classifier and the detector. It's important to note that the first 10,000 images of the table belong to the original dataset. Consequently, it is anticipated that the detector will not detect drift during the initial 10,000 inferences. However, the detection process is expected to be triggered after processing the next batch of 1,000 images, as they are part of the corrupted dataset. Subsequently, the inference system will automatically initiate the training AIFs via the MTO for retraining as shown in Figure 90. Following the initiation of the training AIFs, the script proceeds to make

inferences on the remaining 9,000 images using the newly updated version of the model and detector. Throughout this process, the accuracy metrics are consistently updated and stored in the "CIFAR10Validation" table. This updated accuracy data is then ready for visualization and plotting, providing insights into the model's performance after the retraining phase.

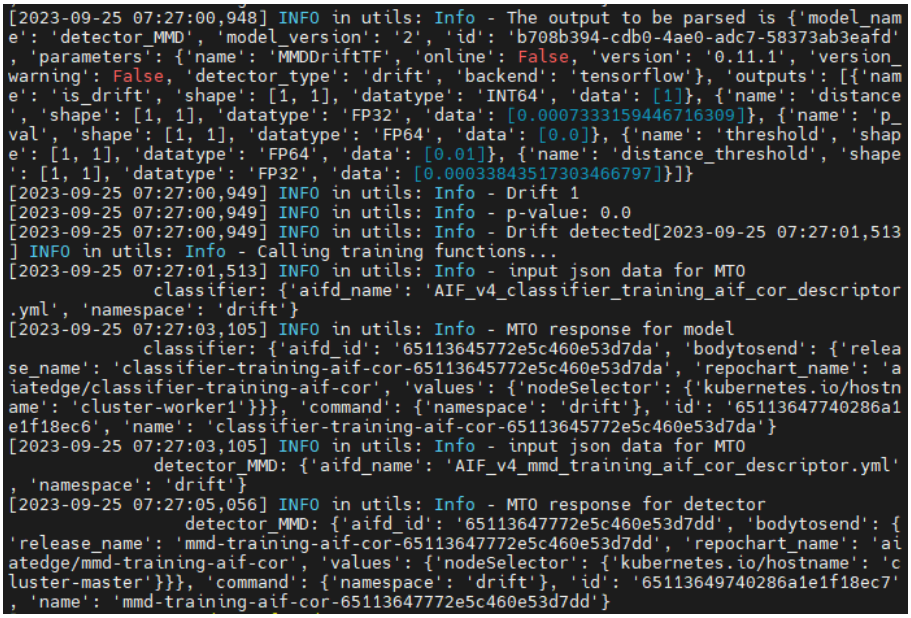


Figure 90 Inference system log when a drift case is detected. The MTO response is shown as well

Results:

- **Detector**: The drift detection process relies on statistical methods, making it challenging to precisely determine the optimal number of samples required at the detector's input to yield reliable results and minimize unnecessary model and detector retraining. There exists a trade-off between the input batch size and detection accuracy. Smaller sample sizes offer less statistical power for accurate drift detection, while larger batch sizes result in longer model operation with potentially reduced accuracy. To address this challenge, we conducted empirical adjustments to the sample size based on observed outcomes. The detector was ultimately initialized with the following parameters: a significance level (p-value) of 0.01, an input batch size of 1,000, and training data consisting of 10,000 images from the training dataset. To validate the performance of this detector configuration, inference was conducted using the test dataset, which included both original and corrupted images. With a batch size of 1,000, this process yielded a total of 20 output values, which are presented in Table 24.

Table 24 MMD detector test for batch size of 1000

	p-value	Drift
Original dataset	0.18	No
	0.15	No
	0.24	No
	0.08	No
	0.93	No
	0.32	No
	0.26	No
	0.49	No
	0.12	No
	0.37	No
Corrupted dataset	0.0	Yes
	0.0	Yes
	0.0	Yes
	0.0	Yes
	0.0	Yes
	0.0	Yes
	0.0	Yes
	0.0	Yes
	0.0	Yes
	0.0	Yes

Although a more fine-grained test can be done to determine a batch size that maximizes the performance of the detector, for the purpose of validating this scenario, a batch size of 1,000 suffices to demonstrate how the detector effectively identifies drift and triggers retraining. However, it is important to note that when the same data is repeatedly injected into the detector, it is possible to obtain slightly different p-values, which can lead to variations in the significance level. As a result, the certainty of drift detection is not absolute. Additionally, the ideal batch size can vary depending on the specific characteristics of the data, the complexity of the detector, and computational resources.

- **Model accuracy results:** As has already been explained in previous subsections, the result of the scenario validation consists of two different graphs created with Grafana using MySQL as data source. These graphs are plotted against the “Accuracy” column from the “CIFAR10Validation” table (y-axis) and an identifier (Id) on the x-axis, where accuracy values oscillate between 0 and 1.

Figure 91 Accuracy of model that has not been retrained

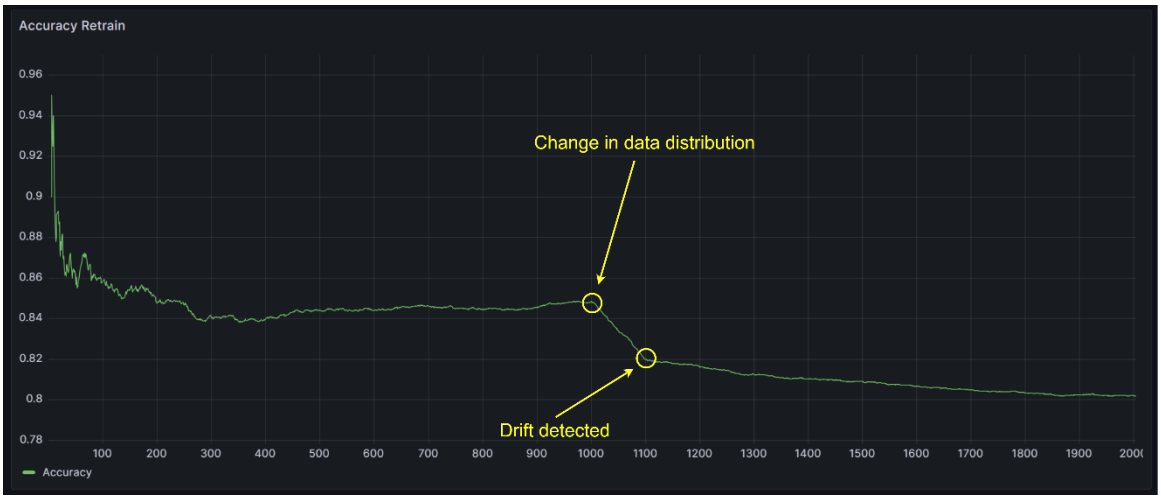


Figure 92 Accuracy of model with retraining

In Figure 91 we observe that when the input data fed into the model is not being actively monitored, the accuracy level experiences a sharp decline, falling below 0.7 as soon as inference begins with the corrupted data. This drop indicates the model's performance degradation when faced with corrupted inputs. In Figure 92, we see a similar trend as in Figure 91. This similarity arises because the same dataset and model are employed for inference. The significant divergence between the two graphs occurs when drift is detected, prompting the initiation of the retraining process. Beyond Id 1100 (when 1,000 available corrupted images are sent to the detector), the model is retrained with the corrupted training dataset, followed by deployment to MLFlow. After the retraining process, which is triggered by drift detection, the accuracy values in Figure 92, show significant improvement compared to the model without monitoring. These values stabilize at approximately 0.8 during the last 900 inferences. This enhancement demonstrates the effectiveness of

monitoring and retraining in maintaining and potentially improving model performance in the face of data drift.

Concluding Remarks on the Validation of Scenario9

The results obtained in this last scenario testing indicate that the implementation of a drift monitoring system yields improvements in the QoS for a Machine Learning model deployed within the AI@EDGE architecture. Nevertheless, several challenges emerge in developing a reliable drift detector for the model, including considerations related to the model itself, data distribution, and batch size, among others. Therefore, a comprehensive analysis is essential to mitigate the risk of deploying a detector that produces unreliable results. Furthermore, it is crucial to account for the time required for inference with the detector and model. In the MMAIF architecture presented here, the total latency for inference depends mainly on the batch size selected. For instance, with a batch size of 1000 samples, the inference process takes 110 seconds. Removing the inference system reduces the total time to 102 seconds. Similarly, reintroducing the inference system and conducting inference directly within the MEC system maintains the total time at 102 seconds. Thus, the inference system and network introduce an additional 8 seconds (7% of the total time) each, excluding those components that contribute the most significant latency. Therefore, there exist a trade-off between the batch size and the time required for the MLServer to perform inference on the model and detector. This trade-off may pose challenges in use cases where rapid model and detector response times are critical requirements.

5 Conclusions & Future Directions

The current deliverable D4.2 presented the developments in the entire AI@EDGE CCP platform following the design and initial development of CCP presented in D4.1. Furthermore, as its main purpose, D4.2 reported the validation methodologies/testbeds & performance results for the CCP components and its integrated version.

D4.2 started by describing the WP4 work performed to extend multiple components/features of the Connect-Compute Platform of D4.1: The usage of *Serverless in* CCP both for more general scenarios (with Managers in LightEdge and Nuclio) and for AI-specific life-cycle management (Model serving, monitoring, and update), *Acceleration* in CCP with HW-SW integration and representative coding & containerization, *Cross-Layer Multi-Connectivity* in CCP with MPTCP integration in the testbed & UC4 devices, *Integration* of the entire CCP with architecture consolidation and example implementation of MTO-MEO-MECPM interfaces. Following the approach proposed in D4.1, the initial AIF descriptor specification has been refined to model various AI-related aspects relevant for the management of the AIF life-cycle. This specification is then used directly by the CCP components for, e.g., resource- and acceleration-aware orchestration, monitoring, autoconfiguration, and update.

The document continued in Section 3 to describe the reference testbed of AI@EDGE (at FBK premises), which comprises of a Kubernetes cluster spanning both near and far edge nodes, already utilizing Athonet's 5G Core, LightEdge and Nuclio Platforms. In addition, it described an auxiliary testbed facilitating edge computing acceleration (at ICCS site), which comprises of multiple servers and diverse FPGA & GPU devices. The validation methodology in Section 3 lays down multiple distinct tests grouped in component-

specific "scenarios", which were devised to evaluate each CCP feature, individually, as well as to assess the entire integrated CCP. The results of Section 4 showed that the aforementioned CCP components all function properly and provide promising performance metrics.

More specifically, regarding the CCP evaluation, the **results of acceleration** showed that we can have an order of magnitude improvement in the compute capabilities of the Cloud-Edge continuum when we strategically place accelerators in the MEC system and perform effective resource management (with the proposed IARM showing ability to host AI/ML algorithms and assist the MEO component). The **results of multi-connectivity** tests showed that, the proposed technology based on multipath aggregation can double the bitrate, while maintaining a near 100% availability. The predictive scheduler systems, built on top of an xApp and an up-to-date OpenRAN SRS stack with an operational E2 interface, allows maintaining connection performance upon interface or backhauling link failure or degradation. An important advantage of the proposed technology is the fact that it is an over-the-top solution, not modifying physical, data-link and network layer, acting instead at the transport layer using multipath Transmission Control Protocol. The **evaluation of the usage of AI-specific Serverless solutions** (e.g., Seldon Core, the sidecar architecture for monitoring and update) demonstrates its benefits for the management of the AIF life-cycle. While with respect to the deployment of pre-built solutions an overhead is introduced, in the zero-touch automation scenarios (where e.g., the ML models are being created continuously) such approaches allow for continuous and autonomous updates of the deployed AIFs without redeployment and downtime. The **evaluation of the orchestration** subsystem and its integration with NSAP, in the end, demonstrated robustness and efficiency of the implemented components, as well as of their integration in different scenarios and setups, such as standalone and non-standalone modes, hybrid MEC systems with different types of resources, and across different MEC systems. The **evaluation of auto-configuration of AIF** through meta-learning has demonstrated its effectiveness compared to conventional solutions and allows for the maintenance of performance with each deployment or relearning. The **evaluation of the non-RT RIC** showed how rApps can implement intelligent control loops by exploiting the exposed telemetry and RAN control services; in particular, we demonstrated an rApp which dynamically manages the radio resources assigned to multi-RAT slices. Finally, the model monitoring scenario underscored that the process of retraining models in response to instances of covariate drift, can effectively mitigate the observed degradation in performance over time (despite considerable challenges that will be tackled via further research in the area).

Overall, the CCP was presented as an extension of ETSI MEC and ETSI MEC in NFV architectures for multi-site and distributed environments; as described in this deliverable, the CCP already includes crucial features in this direction with consortium having devised custom tests & methodologies, which validated its functionality. In future directions, the CCP work can continue towards the inclusion of more applications and models, which will enhance the AI@EDGE platform with all the context and metadata necessary to improve its automatic actionable decisions and to realize intelligent data and computation offload control and management of applications and services.

Bibliography

[1] AI@EDGE deliverable D4.1 “Design & initial prototype of AI@EDGE Connect-Compute Platform” (H2020-ICT-52-2020)

[2] AI@EDGE deliverable D3.1 “Initial Report of systems and methods for AI@EDGE platform automation” (H2020-ICT-52-2020)

[3] AI@EDGE deliverable D3.2 “Final Report of systems and methods for AI@EDGE platform automation” (H2020-ICT-52-2020)

[4] ETSI, "Network Functions Virtualisation (NFV); Management and Orchestration; Os-Ma-Nfvo reference point - Interface and Information Model Specification", ETSI, GS NFV-IFA 013, V2.1.1, October, 2016.

[5] ETSI, “Multi-access Edge Computing (MEC); Framework and Reference Architecture”, ETSI GS MEC 003 V2.2.1 (2020-12)

[6] Open source SDR 4G/5G software suite from Software Radio Systems (SRS), https://www.srslte.com

[7] Open source project of 5GC and EPC, https://open5gs.org

[8] Kubernetes, https://kubernetes.io

[9] Diamanti, Alessio and Vílchez, José Manuel Sánchez and Secci, Stefano, An AI-empowered framework for cross-layer softwarized infrastructure state assessment, IEEE Transactions on Network and Service Management, 2022

[10] Kubernetes plugins, https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/

[11] NVIDIA K8s plugin, https://github.com/NVIDIA/k8s-device-plugin

[12] Xilinx K8s plugin, https://github.com/Xilinx/FPGA_as_a_Service/tree/master/k8s-fpga-device-%20plugin

[13] Blaise et al. “Detection of zero-day attacks: An unsupervised port-based approach”. In: Computer Networks 180 (2020), hal-02889708.

[14] Patetta et al. “A Lightweight Southbound Interface for Standalone P4-NetFPGA SmartNICs”. In: Proceedings of 6GNET (2022), demo hal-03702720.

[15] ZeroTier – Global Area Networking, https://www.zerotier.com/

[16] ETSI, “Multi-access Edge Computing (MEC)}; Federation enablement APIs”. ETSI GS MEC 040 V3.1.1, February 2023

[17] Docker (2023). [Online] Available at: https://www.docker.com/ (Accessed: 14 July 2023)

[18] Helm: The package manager for Kubernetes (2023). [Online] Available at: https://helm.sh/ (Accessed: 14 July 2023)

[19] MLServer: An open source inference server for your machine learning models (2021). [Online] Available at: https://mlserver.readthedocs.io/en/stable/ (Accessed: 14 July 2023)

[20] Seldon Core (2023). [Online] Available at: https://www.seldon.io/solutions/open-source-projects/core (Accessed: 14 July 2023)

[21] Alibi detect: Open source Python library focused on outlier, adversarial and drift detection (2023). [Online] Available at: https://www.seldon.io/solutions/open-source-projects/alibi-detect/ (Accessed: 14 July 2023)

[22] MinIO: High performance object storage for AI (2023). [Online] Available at: https://min.io/ (Accessed: 14 July 2023)

[23] Flask (2023). [Online] Available at: https://flask.palletsprojects.com/en/2.3.x/ (Accessed: 14 July 2023)

[24] Joshi and Jain. "Security and Privacy Considerations for Hardware Accelerators in Cloud and Edge Computing".

[25] Pottie and Kaiser. "Security and Privacy in Hardware Accelerators for Cloud and Edge Computing".

[26] Ghodke. "An Overview — Systems-On-Chips (SOCs) And Their Security Risks," 2021.

[27] Wang, Zhang, and Li. "Securing Hardware Accelerators in Cloud and Edge Computing".

[28] Neumann and Francillon "Confidentiality issues on a GPU in a virtualized environment." In International Conference on Financial Cryptography and Data Security, p. 119–135, 2014.

[29] Jiang, Fei, and Kaeli. "A complete key recovery timing attack on a GPU." In: High Performance Computer Architecture (HPCA), p. 394–405, 2016.

[30] Danopoulos, Stamoulias, Lentaris, Masouros, Kanaropoulos, Kakolyris, and Soudris. "LSTM Acceleration with FPGA and GPU Devices for Edge Computing Applications in B5G MEC." In: Int'l Conf. on Embedded Computer Systems, pp. 406-419. Springer, 2022.

[31] ETSI GS MEC 010-2 [GS MEC 010-2 - V2.2.1 - Multi-access Edge Computing (MEC); MEC Management; Part 2: Application lifecycle, rules and requirements management (etsi.org)]. Mobile Edge Computing (MEC), ETSI Group Specification MEC 010.

[32] Anser, François, and Chrisment, "Auto-tuning of Hyper-parameters for Detecting Network Intrusions via Meta-learning." In: NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium, Miami, FL, USA, 2023, pp. 1-6, doi: 10.1109/NOMS56928.2023.10154381.

[33] Betzler, Camps-Mur, and Catalan. "G-ADRR: Network-Wide Slicing of Wi-Fi Networks with Variable Loads in Space and Time." In: *IEEE Transactions on Mobile Computing*, vol. 21, no. 11, pp. 3986-4000, 1 Nov. 2022, doi: 10.1109/TMC.2021.3066875.

[34] Høiland-Jørgensen, Hurtig, and Brunstrom. "PoliFi: Airtime Policy Enforcement for WiFi." In: *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, Marrakesh, Morocco, 2019, pp. 1-6, doi: 10.1109/WCNC.2019.8885440.

[35] O-RAN Alliance, "O-RAN Use Cases Detailed Specification 11.0", O-RAN.WG1.Use-Cases-Detailed-Specification-R003-v11.00, June 2023.

Annex 1: AIF Descriptors

AIF Descriptor Specification

Base AIF metadata description

AppD ETSI MEC010-2, ETSI MEC037 (draft, waiting for release)

GS MEC 010-2 - V2.2.1 - xMulti-access Edge Computing (MEC); MEC Management; Part 2: Application lifecycle, rules and requirements management (etsi.org)

The base AIF descriptor relies on the attributes of the MEC AppD. Note, however, a definition change with respect to the original model related to virtualComputeDescriptor and swImageDescriptor. In the AIF descriptor these attributes are optional as it is foreseen the possibility to define multiple deployment models for the same AIF.

Attribute name	Card.	Data type	Description	
appDId	1	String	Identifier of this MEC application descriptor. This attribute shall be globally unique. The appDId shall be used as the unique identifier of the application package that contains this AppD An AIF will be rappresented by one appDId. (MEC application attributes applies to AIFs)	NOTE 1
appName	1	String	Name to identify the MEC application	
appProvider	1	String	Provider of the application and of the AppD	
appSoftVersion	1	String	Identifies the version of software of the MEC application	
appDVersion	1	String	Identifies the version of the application descriptor.	
mecVersion	1	String	Identifies version(s) of MEC system compatible with the MEC application described in this version of the AppD. The value shall be formatted as comma-separated list of strings. Each entry shall have the format <x>.<y>.<z> where <x>, <y> and <z> are decimal numbers representing the version of the present document. Whitespace between list entries shall be trimmed before validation. REQUIRED	NOTE 2

appInfoName	0..1	String	Human readable name for the MEC application.	NOTE 1
appDescription	1	String	Human readable description of the MEC application.	
kdu	1	kdu	Kubernetes deployment unit.	
AIFDescriptor	1	AIFDescriptor	AIF specific descriptor	
appServiceRequired	0..N	ServiceDependency	Describes services a MEC application requires to run. NOT REQUIRED Attributes of ServiceDependency	NOTE 2
appServiceOptional	0..N	ServiceDependency	Describes services a MEC application may use if available. NOT REQUIRED Attributes of ServiceDependency (as previous attribute)	
appServiceProduced	0..N	ServiceDescriptor	Describes services a MEC application is able to produce to the platform or other MEC applications. Only relevant for service-producing apps. NOT REQUIRED	
appFeatureRequired	0..N	FeatureDependency	Describes features a MEC application requires to run. NOT REQUIRED [feature name, version]	
appFeatureOptional	0..N	FeatureDependency	Describes features a MEC application may use if available. NOT REQUIRED [feature name, version]	
transportDependencies	0..N	TransportDependency	Transports, if any, that this application requires to be provided by the platform. These transports will be used by the application to deliver services provided by this application. Only relevant for service-producing apps. This attribute indicates groups of transport bindings which a service-producing MEC application requires to be supported by the	

			platform in order to be able to produce its services. At least one of the indicated groups needs to be supported to fulfil the requirements. NOT REQUIRED
appTrafficRule	0..N	TrafficRuleDescriptor	Describes traffic rules the MEC application requires. NOT REQUIRED
appDNSRule	0..N	DNSRuleDescriptor	Describes DNS rules the MEC application requires. NOT REQUIRED Attributes of DNSRuleDescriptor NOT REQUIRED
appLatency	0..1	LatencyDescriptor	Describes the maximum latency tolerated by the MEC application. NOT REQUIRED The value of the maximum latency in nano seconds tolerated by the MEC application. The latency is considered to be the one way end-to-end latency between the client application (e.g. in a device) and the service (i.e. the MEC application instance).
terminateAppInstanceOpConfig	0..1	TerminateAppInstanceOpConfig	Configuration parameters for the Terminate application instance operation. follow the definition in clause 7.1.5.7 of ETSI GS NFV IFA 011 NOT REQUIRED
changeAppInstanceStateOpConfig	0..1	ChangeAppInstanceStateOpConfig	Configuration parameters for the change application instance state operation. follow the definition in clause 7.1.5.8 of ETSI GS NFV IFA 011 NOT REQUIRED

userContextTransferCapability	0..1	UserContextTransferCapability	If the application supports the user context transfer capability, this attribute shall be included. NOT REQUIRED	NOTE 2
appNetworkPolicy	0..1	AppNetworkPolicy	If present, it represents the application network policy of carrying the application traffic. NOT REQUIRED	
NOTE 1: Content same for MEC and NFV, inherit from TOSCA type defined in ETSI GS NFV-SOL 001 [3] NOTE 2: Content as in ETSI GS MEC 010-2				

Deployment Profiles

The deployment profiles section defines the potential LCM configurations for the same function.

Attribute name	Card.	Data type	Description
kdu	0..N	KDUType	Deployment profile definition

Each deployment profile is defined with the following Kubernetes Deployment Unit structure1.

Attribute name	Card.	Data type	Description
name	1	String	Profile name. Unique at the level of the same AIF definition. Must confrom helm deployment name standard.
description	0..1	String	Optional profile description.
helm-chart	1	String	Reference to the Helm chart definition. Uniquely identifies the Helm chart in the chart repository. The chart repository must be remote.
requirements	1	DeploymentRequirementsType	Deployment requirements. Consists of k8s-cluster (0..1): Information about Kubernetes cluster requirements (version, cni, nets) virtualComputeDescriptor[2] (required): CPU and memory requirements. virtualStorageDescriptor[3] (0..N): Persistent storage requirements.

			accelerationDescritor (0..1): Definition of acceleration capabilities required for this profiles
metrics	0..N	MetricsType	Advertised metrics for this deployment. Defines a list of metrics, each with id: metric identifier minValue (optional): min value of the metric provided with this profile maxValue (optional): max value of the metric provided with this profile

Requirements Definition: DeploymentRequirementType

The requirements definition consists of k8s cluster requirements, computation descriptor, storage descriptor and acceleration descriptor.

K8S Cluster Requirements[4]

Attribute name	Card.	Data type	Description
version	1	String	Kubernetes versions supported, comma-separated.
cni	0..1	String	Container Network Interface plugin to be used. NOT REQUIRED
nets	0..1	NetworkType	List of network links represented with id

Computation requirements: **VirtualComputeDescriptor**[5]

Attribute name	Card.	Data type	Description
virtualCpu / cpuArchitecture	0..1	String	CPU architecture type (e.g., x86, ARM)

virtualCpu / numVirtualCpu	1	Number	Number of Virtual CPUs
virtualCpu / virtualCpuClock	0..1	Number	Minimum CPU clock rate (in MHz)
virtualCpu / virtualCpuPinning / virtualCpuPinningPolicy	0..1	Enum	Indicates the policy for CPU pinning. VALUES: STATIC DYNAMIC In case of "STATIC" the virtual CPU cores are requested to be allocated to logical CPU cores according to the rules defined in virtualCpuPinningRules. In case of "DYNAMIC" the allocation of virtual CPU cores to logical CPU cores is decided by the VIM (e.g. SMT (Simultaneous Multi-Threading) requirements).
virtualCpu / vduCpuRequirements	0..N	Not Specified	Array of key-value pair requirements on the Compute (CPU) for the KDU.
virtualMemory / virtualMemSize	1	Number	Amount of virtual Memory (e.g. in MB).
virutalMemory / vduMemRequirements	0..N	Not Specified	Array of key-value pair requirements on the memory for the KDU.
virtualDisk / sizeOfStorage	0..1	Number	Size of the storage (in MB)
virtualDisk / vduStorageRequirements	0..N	Not Specified	Array of key-value pair requirements on the disk for the KDU.

Storage Requirements: VirtualStorageDescriptor[6]

Attribute name	Card.	Data type	Description
typeOfStorage	1	Enum	Type of virtualised storage resource. VALUES: BLOCK OBJECT FILE
blockStorageData / sizeOfStorage	0..1	Number	Size of the storage (in MB). ONLY IF typeOfStorage = BLOCK
blockStorageData / vduStorageRequirements	0..N	Not Specified	Array of key-value pair requirements on the disk for the KDU. ONLY IF typeOfStorage = BLOCK Cardinality
objectStorageData / maxSizeOfStorage	0..N	Number	Max size of virtualised storage resource in GB. ONLY IF typeOfStorage = OBJECT
fileStorageData / sizeOfStorage	0..N	Number	Size of virtualised storage resource in GB. ONLY IF typeOfStorage = FILE
fileStorageData / fileSystemProtocol	0..N	String	The shared file system protocol (e.g, NFS, CIFS). ONLY IF typeOfStorage = FILE
perInstance	0..1	Boolean	Indicates whether the virtual storage resource shall be instantiated per instance. If the value is true (default), a virtual storage resource shall be instantiated for each instance that is based on a KDU referring to this virtual storage descriptor and have the same lifetime as the instance. If the value is false, a single virtual storage resource shall be instantiated with a lifetime independent of the lifetime of individual instances based on a KDU referring to this virtual storage descriptor. The storage resource shall have the same lifetime as the AIF instance.

Accelerator Descriptor

Attribute name	Card.	Data type	Description
hardwareType	1	string	(e.g., FPGA, GPU, TPU, VLIW, CPU)
hardwareDevice	1	string	(e.g., U280, V100, Zynq XX)
hardwareDeviceMemory	1	number	(e.g., 1000 (in MB))
devicesNumber	1	Integer	(e.g., 1,2,3,…)
softwareHostRequirements	0..N	string	(e.g., “Xilinx XRT 2.11 - XRM 1.2 - xdma_201920_3”, “fpga-xilinx_u280_xdma_201920_3-1579649056" or NVIDIA cuda-11.8, “”)

Metrics Descriptor

Metrics descriptor describes the expected KPIs that this deployment profile allows to achieve regarding its output (e.g., data generated, API requests processed, etc).

Attribute name	Card.	Data type	Description
id	1	Identifier	Identifer of the metric (e.g., latency, throughput).
minValue	0..1	Number	Minimum value for the metric guaranteed. Defaults to 0
maxValue	1	Number	Maximum value for the metric guaranteed.

AIF Specific Features

AIF general schema descriptor

General Subdivision for AIF Descriptor

- Predictor
  - AI Model Descriptor
  - Activation
  - Input sources
  - Output
  - Dependencies
- Training
  - AI Model Descriptor
  - Activation
  - Input sources
  - Output
  - Distribution
  - Dependencies

Attribute name	Card.	Data type	Description
AIFDescriptor	0..N	AIFOperation	Specify AIF operation with AIF descriptor

AIF Operation Descriptor

Attribute name	Card.	Data type	Description
operation	1	Enum	Specify AIF operation. VALUES: TRAINING PREDICTION
modelDescriptor	0..1	modelDescriptor	Model info
activation	1	ActivationDescriptor	Trigger for AIF operation
inputSources	0..N	inputSources	Descriptor of AIF input
outputSources	0..N	outputSources	Descriptor of AIF output
dependencies	0..N	dependenciesDescriptor	List of required AIF by this AIF operation
distribution	0..1	distributionStrategy	Descriptor that declares the distribution strategy for a training AIF. Only if operation = TRAINING

Activation descriptor

Event-based triggers activate the function upon the occurrence of an event. There can be three types of trigger events: API-call request, message publication on a channel AIF is subscribed to, activation on deploy.

The time-based triggers activate the function temporally, with onetime or recurring activation, through scheduling or a cron.

Attribute name	Card.	Data type	Description
mode	1	Enum	Modality that triggers AIF. VALUES: • EVENT • TIME • PERMANENT
eventTrigger	0..1	Enum	Event trigger type. VALUES: • API_CALL • MESSAGE_CHANNEL • ON_DEPLOY Required only if triggerMode = EVENT
cron	0..1	String	Cron string that regulates AIF activation. Used only if triggerMode = TIME
interval	0..1	Integer	Seconds between the two consecutive activations. Used only if triggerMode = TIME. Alternative to cron-based trigger
messageChannel	0..1	String	Activating message topic. Used only if triggerMode = MESSAGE_CHANNEL

Input sources descriptor

Attribute name	Card.	Data type	Description
id	1	String	Identifier of data source
description	1	String	Human readable description of data source
dataSourceType	1	Enum	Data source type. VALUES: • **RAW_DATA** • **PIPELINE** • **STORAGE**
dependsOn	0..N	String	Array of data dependencies (other DataSources)
datasourceProperites	1	datasourceProperites	Type-specific properties for the data sources: • For the raw data sources: o **metricRef**: identifier of the metric to collect o **resourceRef**: identifier of the resource generating the metric • For storage data sources: o **storageRef**: URI of the storage to access o **datasetRef**: URI of the dataset relative to the source o **modelRef**: URI of the model to retrieve • For pipeline data sources: o **pipelineRef**: reference to the pipeline (e.g., URI of image or repo, etc.)
dataSize	0..1	Integer	Byte size of data to be transmitted.
dataBandwith	0..1	Integer	The bandwidth required on the transport interface that connects this AIF to the data sources. This depends on the publish frequency and the size of the metrics in the data sources.
dataLatency	0..1	Integer	The latency requirement for the data sources. The values in the data source are only valid if they are consumed by the AIF within a certain time. (So that the data is not stale)

dataWindow	0..1	dataWindow	Identifier of storage where data are stored. Defined as • **timeInterval**: time interval to consider before deployment (e.g., 2h) • **pointInterval**: minimal number of data points available before deployment • **start / end**: timestamps defining explicit time window Either of the above should be provided
dataConfidence	0..N	DataConfidence	If the input to the AIF is the output of another AIF, then confidence intervals associated with them need to be above a certain threshold. Defined as • **propertyRef**: specific property of the data to consider • **value**: threshold value to consider valid
dataAugmentation	0..1	DataAugmentation	Definition of the data augmentation policy. If specified should provide: • **dataAugmentationType**: specific algoritm to apply.
enableDataCollection	0..1	Boolean	Defines whether the data collection should be activated by the orchestrator if the data source is not yet available.
dataAggregation	0..1	DataAggregation	Defines whether the data used by the source should be aggregated (e.g., by extra pipeline operator). Defines: • **dataGranularity**: time intervals to consider for aggregation • **aggregationFunction**: aggregation function to apply (standard or code ref) • **aggregatedProperty**: property to aggregate upon (optional in case of a single property source) • **grouping**: list of grouped properties (optional)

Output descriptor

Attribute name	Card.	Data type	Description
id	1	String	Identifier of data output
description	1	String	Human readable description of data output
outputType	1	Enum	Output data type. VALUES: • TRAINED_MODEL • PREDICTION • PERFORMANCE • CONFIDENCE
outputChannel	1	Enum	Place where output is sent. • STORAGE • MESSAGE_CHANNEL • RAW_DATA

outputProperties	1	any	Output channel properties specific to its type:
			• For storage:
			  ○ **storageRef**: URI of the storage to write to
			  ○ **datasetRef**: URI of the dataset relative to the source
			  ○ **modelRef**: URI of the model
			• For channel data sources:
			  ○ **messageChannel**: id of the channel to write to

outputFormat	0..1	SchemaDef	An optional specification of the output format: JSON Schema, Table Schema, ...

Distribution Strategy descriptor

Attribute name	Card.	Data type	Description
trainingDistrtibutionEnabled	0..1	Bool	Flag to enable distributed learning strategy. Default = False (Which means the training is executed in a centralized manner)
distributionParameter	0..1	distributionParameters	Descriptor to specify how the distributed learning is structured. Only if trainingDistrtibutionEnabled = True

Distribution Parameters descriptor

Attribute name	Card.	Data type	Description
distributionType	1	Enum	Identify the distribution strategy for the AIF: distributed in a parallel computing platform (data distribution) or federated (distributed learning without data sharing). VALUES: • DISTRIBUTED • FEDERATED
distributionMode	0..1	Enum	Distribution modality, client-server or peer-to-peer. VALUES: • CLIENT_SERVER • P2P
clientNumber	0..1	Number	Number of clients required at deployment. Only if distributionStrategy = FEDERATED
minClientAvailable	0..1	Number	Minimum number of client available to perform model update. Only if distributionStrategy = FEDERATED
exchangeChannel	0..1	Enum	Place where output is sent. • STORAGE • MESSAGE_CHANNEL
exchangeChannelProperties	0..1	Any	Output channel properties specific to its type: • For storage: ○ **storageRef**: URI of the storage to write to / read from • For channel data sources: ○ **messageChannel**: id of the channel to write to / read from

In this way, the distribution parameter descriptor represents at high level the i3 interface. The AIF exchanges the worker parameters and/or data through a specific communication channel.

AI Model descriptor

Attribute name	Card.	Data type	Description
modelDebug	0..1	Bool	DebugRun: If set, run step by step for debug purposes at the cost of slower execution time.
modelInfo	1	ModelInfo	Human readable metadata of AIF • modelList: the list of models used • modelInfo: basic information of each of the models, like how many layers and how many parameters. • lossFunction • activationFunction
modelRef	0..1	String	(URI) ref to the pretrained model for the function to work with (both for trained and for predictor)
modeUpdate	0..1	Bool	Whether the function supports dynamic model update
modelUpdateInterface	0..1	ModelUpdateInterface	Specification (API, message, event) for the update trigger interface (?)

Dependency descriptor

Apart from implicit dependencies, it is possible to define also explicit dependencies of AIF with other functions. This may be necessary, e.g., when the prediction function requires the corresponding continuous training to update.

Attribute name	Card.	Data type	Description
AIFDependency	0..N		Each AIF must include a list of other AIFs that it intends to use so that a dependency tree can be built to avoid loops of AIFs with dependency loops. Defined with: • AIF identity as published in the catalogue