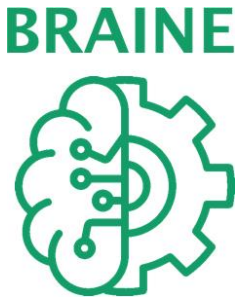

BRAINE - Big data Processing and Artificial Intelligence at the Network Edge

Project Title:	**BRAINE - Big data Processing and Artificial Intelligence at the Network Edge**
Contract No:	876967 – BRAINE
Instrument:	ECSEL Research and Innovation Action
Call:	H2020-ECSEL-2019-2-RIA
Start of project:	1 May 2020
Duration:	36 months

Deliverable No: D3.2

Second report on the status of WP3

Due date of deliverable:	31 March 2022
Actual submission date:	26 April 2022
Version:	1.0

ECSEL Joint Undertaking

Project funded by the European Community under the H2020 Programme for Research and Innovation.

Project ref. number	876967

Project title	BRAINE - Big data Processing and Artificial Intelligence at the Network Edge

Deliverable title	First report on the status of WP3
Deliverable number	D3.2
Deliverable version	Version 1.0
Previous version(s)	-
Contractual date of delivery	31 March 2022
Actual date of delivery	26 April 2022
Deliverable filename	D3.2 Second report on the status of WP3
Nature of deliverable	Report
Dissemination level	PU
Number of pages	64
Work package	WP3
Task(s)	T3.1, T3.2, T3.3, T3.4
Partner responsible	DELL
Author(s)	Mustafa Al-Bado (Dell), Javad Chamanara (LUH), Ahmed Khalid (DELL), Edgard Marx (ECC), Adam Flizikowski (ISW), Janina Habrunner (IFX), Vojtěch Janů (CTU), Ilya Vershkov (MLNX), Radoslav Gerganov (VMware)
Editor	Mustafa Al-Bado (Dell)

Abstract	This technical report, delivers the detailed information about the progresses that have been made in the context of work package 3 (Secure and efficient data management and resource orchestration supporting AI). The report covers architectural and technological designs, decisions, selections, and developments for various aspects of the work package including, Physical Layer Security (PLS) Protocol, Distributed Knowledge Base, Service and Resource Repository, and SLA in edge computing supporting AI. It also covers the advancements regarding the techniques for workload placement and management

	and efficient edge-edge and edge-cloud communication. The report provides implementation details about the telemetry and monitoring developments.
Keywords	

Copyright

Deliverable history

Version	Date	Reason	Revised by
00	01.03.2022	Table of Contents - version 00	Mustafa Al-Bado
01	01.04.2022	All contributions	Mustafa Al-Bado
02	25.04.2022	Ex. Summary and Conclusions	Mustafa Al-Bado
1.0	02.05.2022	Final review	Mustafa Al-Bado, F. Cugini

List of abbreviations and Acronyms

Abbreviation	Meaning
5G	5th Generation
AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
CU	Centralized Unit
DSP	Digital Signal Processors
DU	Distributed Unit
ECG	ElectroCardioGram
EEG	ElectroEncephaloGram
EMDC	Edge Mobile Data Center
EPC	Evolved Packet Core
ERP	Enterprise Resource Planning
EU	European Union
FPGA	Field Programmable Gate Arrays
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
HRC	Human-Robot Collaboration
iDT	intelligent Digital Twin
ICT	Information and Communication Technologies
IP	Internet Protocol
IoMT	Internet of Medical Things
IoT	Internet of Things
IT	Information Technology
KPI	Key Performance Indicator
MES	Manufacturing Execution Systems
MOD	MOtif Discovery
PoC	Proof of Concept
QSD	Qualified Synthetic Data
RAN	Radio Access Network
TBC	To Be Confirmed
TBD	To Be Defined
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TFLOPS	Tera Floating Point Operations Per Second

TSN	Time-Sensitive Networking
UE	User Equipment
URI	Uniform Resource Identifier
URLCC	Ultra-Reliable Low-Latency Communication
USRP	Universal Software Radio Peripheral

Table of Contents

List of Figures

1. Executive summary

Edge computing has increasingly become an integral part of many applications (e.g. mission critical apps), which led to adopting the edge computing concept in several standards in mobile networks (e.g., 5G) and industrial manufacturing. This work package (WP) addresses key aspects of edge networks. Mainly, WP3 covers intelligent allocating, placement, and monitoring workloads in edge environments. This document reports the progress and development in research studies, architecture design, components development in edge resource collaboration, resource management and service deployment, AI/ML-based workload placement, and monitoring infrastructure.

Specifically, the current deliverable highlights the following achievements:

- In Task 3.1, the service mesh component's development is completed to enable cross-cluster communication between BRAINE services. Moreover, a message structure is implemented for the multiagent communication based on FIPA ACL using AMQP protocol. Regarding SDN network controller, the development focused on adding features to the P4 driver to support forwarding of traffic generated among K8s pods and postcard telemetry on traffic flows generated among K8s PODs, in addition to exploring an alternative approach for hierarchical controllers' architecture based on inter-Controller communication. Finally, the efficiency of the physical layer security (PLS) blockchain was investigated.

- In Task 3.2, firstly, new features have been added to the Resource Management & Service Description, such as the vocabulary for Kubernetes service deployment and resource management and interfaces to register applications and services and create service catalogues for deployment. Secondly, MEC platform applications deployment has been implemented and integrated with the BRAINE platform. Finally, SLA broker architecture is designed and implemented for a distributed edge environment.

- In Task 3.3, firstly, several features have been added to the BRAINE scheduler, such as enabling reinforcement learning. Secondly, a cognitive framework is designed to host the platform intelligence and used as-a-service (i.e., cognitive-as-a-service) for the rest of the system. Finally, two AI/ML use-cases are presented in the context of BRAINE about vRAN workload forecasting and prediction and image processing for smart cities applications.

- In Task 3.4, three components have been developed in the network telemetry framework: telemetry monitors, adapters and ingesters. Moreover, the Telegraf agent for 5G data collection has been successfully integrated with the telemetry system.

2. Edge resource collaboration

2.1. Service Mesh

The service mesh component is responsible for building a service mesh between EMDC clusters and allows cross cluster communication between BRAINE services. It exposes a REST API which is defined below. The source code is available at https://gitlab.com/braine/braine-mesh

2.1.1. Adding EMDC cluster

Adds EMDC cluster to the service mesh by uploading k8s config file

```
POST /emdc
```

Parameters are specified in Table 2.1

Table 2.1 Add EMDC parameters

Name	Type	Description
emdc	multipart/form-data	Kubernetes configuration file

2.1.2. Retrieve EMDC cluster

Retrieves the EMDC cluster with the specified id. The "host" field in the response specifies the master node of the EMDC.

```
GET /emdc/{id}
```

Parameters are specified in Table 2.2

Table 2.2 Retrieve EMDC parameters

Name	Type	Description
id	Int	The id of the EMDC

Response:

Status: 200 OK

```
{
  "id":0,
  "host":"https://10.185.99.10:6443"
}
```

2.1.3. Listing all EMDC clusters

Retrieves all EMDC clusters which are registered in the service mesh.

```
GET /emdcs
```

Response:

```
Status: 200 OK
[
  {"id":0,"host":"https://10.185.99.10:6443"},
```

```
  {"id":1,"host":"https://10.78.210.149:6443"}
]
```

2.1.4. Deleting EMDC cluster

Deletes the EMDC cluster with the specified id from the service mesh.

`DELETE /emdc/{id}`

Parameters are specified in Table 2.3

Table 2.3 Delete EMDC parameters

Name	Type	Description
id	Int	The id of the EMDC

2.1.5. Listing all service entries in EMDC

Retrieves all service entry in the EMDC with the specified id.

`GET /emdc/{id}/serviceentries`

Parameters are specified in Table 2.4

Table 2.4 List all service entries parameters

Name	Type	Description
id	int	The id of the EMDC

2.1.6. Creating new service entry

Creates new service entry in the EMDC with id “id1” and the specified namespace. The host should be of the form `<name>.<namespace>.global` where `<name>.<namespace>` is a service running in EMDC with id “id2”.

`POST /emdc/{id1}/serviceentry/create/{id2}`

Parameters are specified in Table 2.5

Table 2.5 Create new service entry parameters

Name	Type	Description
id1	int	The id of the EMDC where the service entry will be created
id2	int	The id of the target EMDC
name	string	The name of the service entry
host	string	The hostname of the service entry
namespace	string	The namespace of the service entry
portnumber	int	The portnumber of the target service

protocol	string	The protocol of the target service

2.1.7. Deleting service entry

Deletes the service entry with the specified uuid in the EMDC with the given id.

```
DELETE /emdc/{id}/serviceentry/{uuid}
```

Parameters are specified in Table 2.6

Table 2.6 Delete service entry parameters

Name	Type	Description
id	int	The id of the EMDC
uuid	string	The uuid of the service entry

2.1.8. Retrieve service entry

Retrieves the service entry with the specified uuid in the EMDC with the given id

```
GET /emdc/{id}/serviceentry/{uuid}
```

Parameters are specified in Table 2.7

Table 2.7 Retrieve service entry parameters

Name	Type	Description
id	int	The id of the EMDC
uuid	string	The uuid of the service entry

Response:

```
Status: 200 OK
{
  "uuid":"ce1ce847-0f1e-40e8-92ac-9922aedf6e55",
  "name":"plot",
  "portnumber":80,
  "protocol":"http",
  "host":"plot.gpu.global",
  "namespace":"x86"
}
```

2.2. Multiagent communication

The P2P communication implements a message structure based on FIPA ACL using AMQP protocol. Each agent is equipped in the northbridge with a communication mechanism that connects as a client to the RabbitMQ server and allows sending

messages to particular agents. The messages for each agent are stored in the agent's queue from which the agent can retrieve the incoming message (See Figure 2.1).

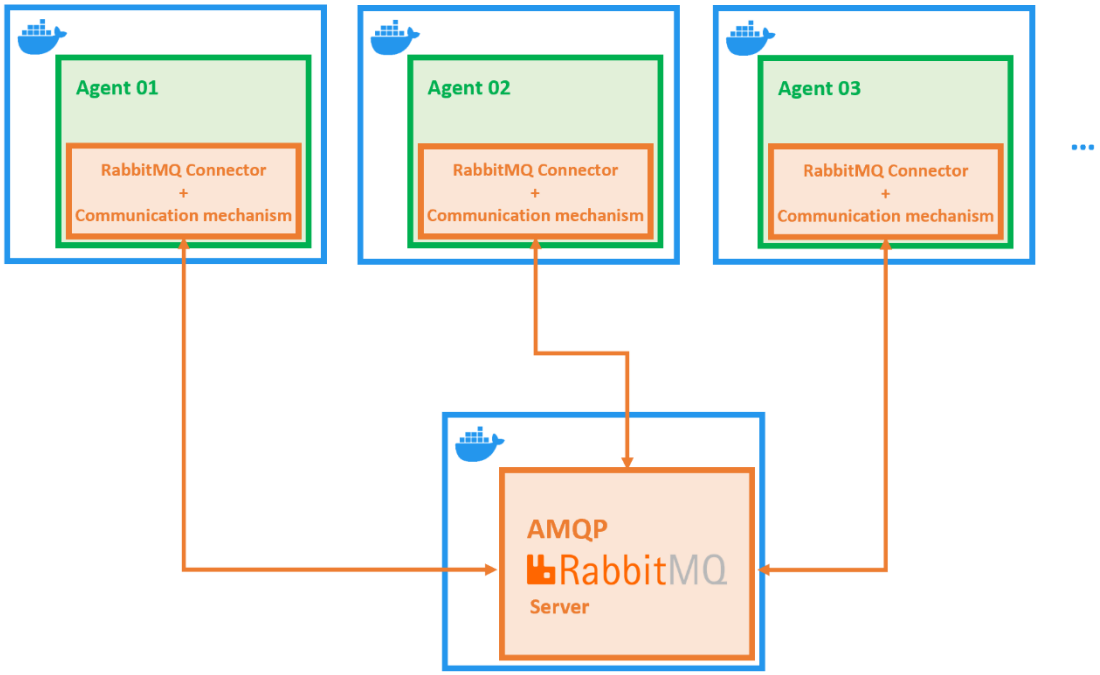

Figure 2.1: Dockerized multiagent architecture from the point of view of the communication

2.2.1. Message Structure

The message structure (Table 2.8) is based on FIPA ACL Message Structure Specification and is inspired by HTTP headers (RFC 6648, RFC 4229). The structure was designed to give precise information about the current state and intentions of agents participating in the conversation.

Table 2.8: Message fields of multi-agent communication

Name	Based on	Mandatory	Description
Sender	FIPA	Yes	Denotes the identity of the sender of the message
Receiver	FIPA	Yes	Denotes the identity of the intended recipients of the message
Ontology	FIPA	No	Denotes the ontology(s) or other knowledge structure used to give meaning to the symbols in the content expression
Message ID	Custom	Yes	Used to identify the particular message in a conversation
In reply to Message ID	FIPA	No	Denotes an expression that references an earlier action to which this message is a reply
Reply to	FIPA	No	This parameter indicates that subsequent messages in this conversation thread are to be

			directed to the agent named in the `reply-to` parameter instead of to the agent named in the `sender` parameter
Conversation ID	FIPA	No	Introduces an expression (a conversation identifier) that is used to identify the ongoing sequence of communicative acts
Performative	FIPA	Yes	Denotes the type of the communicative act of the message
Communication protocol	FIPA	No	Denotes the interaction protocol that the sending agent is employing
Content format	HTTP	Yes	Content type expressed as MIME type
Content encoding	FIPA	No	Denotes the specific encoding of the content
Timestamp	HTTP	No	In HTTP called `date`. Contains the date and time at which the message was originated.
Authorization		No	It is used to provide credentials that authenticate an agent.
Accept content format	HTTP	No	Denotes content types that the agent understands. An example is `text/json`
Authenticate	HTTP	No	Authentication methods ("challenges") might be used to authenticate an agent. In HTTP `www-authenticate`
Expires	FIPA	No	Denotes a time and/or date expression which indicates the latest time by which the sending agent would like to receive a reply. In FIPA called `reply-by`
Accept encoding	HTTP	No	Denotes the content encoding that the agent can understand
Cors	HTTP	No	Denotes desire to block cross-origin\cross-tenant communication
Content	FIPA	Yes	Denotes the content of the message; equivalently denotes the object of the action. The meaning of the content of any ACL message is intended to be interpreted by the receiver of the message.

2.2.2. Communication infrastructure setup

The format for the multiagent communication uses used AMQP protocol, and the RabbitMQ is used as a broker. From the AMQP model, a message queue is bound to each agent, and for the exchange is used the direct exchange that allows P2P messaging. Inside the agents, the communication mechanism uses asynchronous

message processing based on asynchronous execution support in the Spring framework, built on the top of the Join/Fork framework.

2.3. SDN network controller

The ONOS SDN controller has to be deployed in each EMDC. Figure 2.2 reports the main components that we have specifically deployed for the BRAINE project.

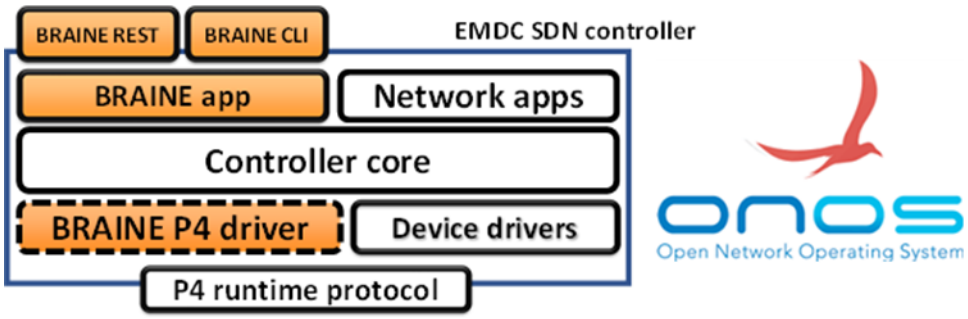

Figure 2.2: main components of the BRAINE SDN controller.

The components detailed below were developed in the previous reporting period, and deeply described in the D3.1 document.

- The REST APIs library on the north-bound interface to interact with the EMDC orchestrator is fully functional (e.g., to receive request for the configuration of a new connectivity).
- The CLI command library on the north-bound interface to interact with human users. This interface is especially useful during development for testing purpose.
- The BRAINE application that utilizes the ONOS core services to deploy the requests received from the REST APIs and the CLI commands.

In the current period the development work has focused on the P4 driver that has been extended to connect and install flow rules on P4 devices for supporting: (i) forwarding of traffic generated among K8s PODs; (ii) postcard telemetry on traffic flows generated among K8s PODs.

Moreover, several upgrades to the REST and CLI interfaces have been applied for enabling the integration toward the K8s orchestrator and other BRAINE components, such as the SLA broker and the Telemetry system.

Finally, a deep testing/debugging campaign has been accomplished and the ONOS BRAINE controller has been demonstrated live in the OFC conference in March 2022. More details are reported in Section 2.3.1.

The software component described above have been released in two software package that can be dynamically installed on a running ONOS controller. The current version of the software package is available at:

- https://gitlab.com/braine/WP3-SDN-CONTROLLER
- https://gitlab.com/braine/wp3-sdn-controller-p4

Moreover, advanced hierarchical architectures of ONOS controllers have been studied to operate in multi EMDC scenarios allowing the control of multi-layer networks including hybrid packet/optical nodes.

2.3.1. Extended In-Band Telemetry for Monitoring-Driven Traffic Steering

This section reports on a first implementation of P4-based telemetry tool (i.e., inband telemetry). This work represents an important step toward the implementation of the BRAINE EMDC architecture illustrated in Figure 2.3, but is focused on the telemetry implementation at the P4-device level, i.e., it is not configurable by the SDN controller. The work progress in this direction, i.e., enabling the SDN control of P4-based telemetry is ongoing and is based on the SDN controller application reported in the previous section. A first set of results regarding the control of telemetry by the SDN controller will be reported in D5.4, including a detailed description of the integration of the SDN controller with other EMDC components.

The P4-based EMDC solution reported in this section support in-band telemetry (INT). INT is a specifically designed header that can be added/modified/removed, reporting useful metadata such as the time spent in the outgoing queue [Paol]. The analysis of retrieved INT data on accumulated delay can drive innovative dynamic packet scheduling solutions (e.g., dynamic per-packet classification and priority enforcement), minimizing jitter and maximum experienced end-to-end latency. Furthermore, INT data could be exploited to derive long term statistics of latency/service performance across the whole network, potentially leading to global network re-optimizations to be enforced by the SDN Controller, possibly leveraging on AI-based adaptive strategies.

A relevant use-case to exploit INT-based solutions at the edge is to enable the user equipment (UE) of a 5G network to directly enforce to its outgoing packets the INT extra-header (e.g., relying on an embedded P4 software implementation). Indeed, monitoring the actual performance (e.g., latency) of selected applications across the whole e2e path is often not possible given the presence of multiple providers, roaming between 5G Operators, edge and cloud providers, and transport network operator(s). On the other hand, extending INT from the UE enables accurate latency monitoring of the whole e2e path, including both the wireless and wired segments, even if operated by different service providers. Other important used cases includes 5G offloading and decentralized cybersecurity [Cugi].

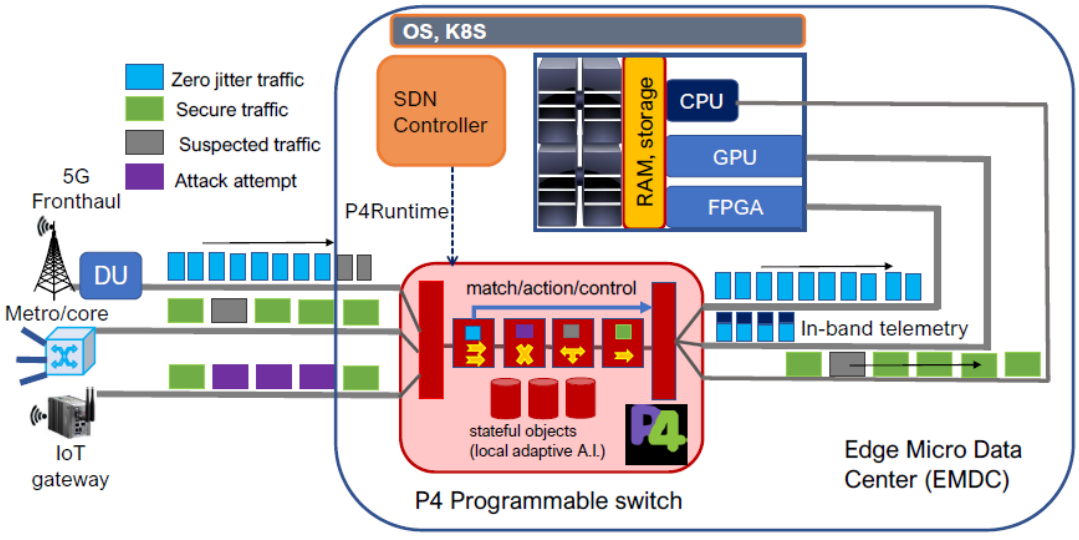

Figure 2.3: BRAINE EMDC architecture.

The EMDC architecture including both heterogenous computing and programmable networking resources used for monitoring traffic, 5G offloading, and decentralized cyber-security is illustrated in Figure 2.3.

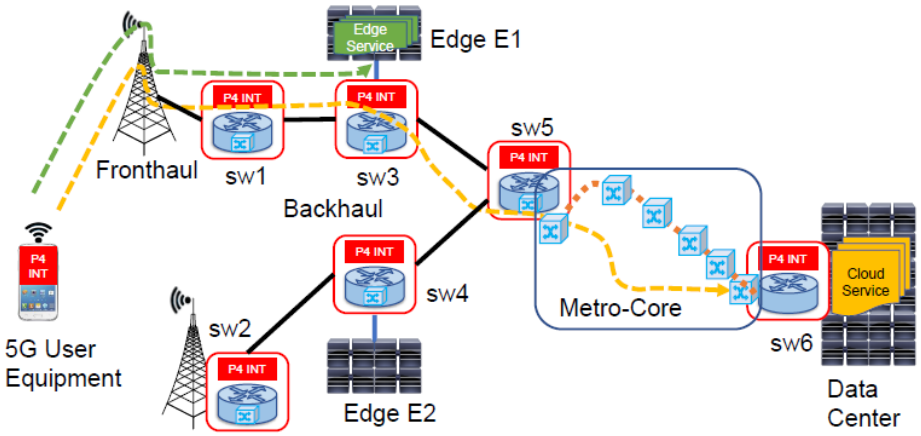

Figure 2.4: Scenario of 5G network served by backhaul and metro-core network with cloud and edge resources.

The considered network scenario highlighting the potential of the UE-based INT solution is shown in Figure 2.4. The UE is connected to the cloud through a network encompassing the 5G network, the 5G backhaul and the metro-core network segments. Edge nodes (i.e., E1 and E2) are located at the backhaul segment, which is composed of programmable forwarding elements (i.e., sw1-sw6 switches) supporting INT. INT is used to collect the time spent in queue by each traversed switch. Service applications can run either in the cloud or in the edge computing node, based on the latency requirements. Typically, INT is programmed to monitor just the wired segment. However, in a wired-wireless e2e path, significant link latency variations may occur, particularly in the Radio Access Network (RAN) system segment, traditionally not monitored by INT. Indeed, latency variations due to mobility, users' subscription, queuing delay, frame alignment and transmission processing may heavily affect service performance.

To enable e2e INT-based monitoring and dynamic cloud edge steering without involving the controller, three main technologies need to be developed: a) UE inclusion in the

INT domain, b) extended handling of the INT Report packet to compute link latency and c) automatic source-based edge-cloud steering. First, a programmable switch is implemented within the UE as a software service app (e.g., a lightweight virtual container) and programmed to act as INT source node. Transit nodes, including the Edge switch, update the INT values. The destination node, e.g., the cloud gateway, removes all INT headers providing traffic transparently to the server. In addition, the destination, instead of sending the INT Report message to the Controller, it forwards the Report packet in the backward direction up to the UE. Also, the edge node is configured to pop INT headers and send Report messages in case the traffic is steered at the Edge without reaching the cloud. The Report packet is also used to collect latency information in the backward direction from the cloud to the UE. This way, it is possible to correlate timestamp information at each traversed node as well as monitor the whole bidirectional e2e latency performance, also including the wireless link. The experienced latency of the latest N packets is stored at the UE in a P4 register.

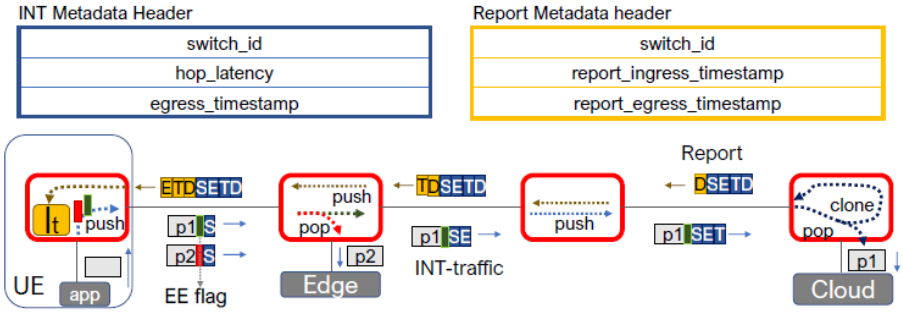

Figure 2.5: In-band telemetry at the user equipment for latency monitoring and automatic decentralized steering.

Finally, INT is also augmented to include extra fields. A specifically added flag called EnableEdge EE is set by the UE if the experienced latency is not satisfactory (Figure 2.5). If the switch at the edge detects the EE flag set, it triggers the steering at the edge. This way, the UE performs source-based steering imposing traffic to reach the closer edge in case latency threshold is exceeded, without involving controllers. To avoid instabilities, EE is activated when a pre-determined number of packets exceeds the threshold, utilizing the aforementioned stateful capability of the P4 technology.

2.3.2. Peer collaboration of SDN controllers

This section explores one possible solution to coordinate multiple SDN controllers to operate on a packet/optical network including P4-based packet devices, hybrid packet/optical devices, and traditional optical devices.

Disaggregated optical networks have attracted remarkable interest due to potential savings in CapEx as well as for their fully standardized open interfaces for SDN [Ricc, Chon, Hern]. In this context, most of the scientific work on disaggregation has focused on optical transmission modules as standalone network elements, like transponders and muxponders. However, the recent advances in transmission technology have driven the introduction of coherent pluggable transceivers that can be equipped within packet switching devices. For example, Digital Coherent Optics (DCO) transceivers are commercially available at rates of 400 Gbps with configurable transmission parameters in different form factors, such as CFP2 and the smaller QSFP-DD 400ZR. Replacement of standalone transponders with pluggables modules in the packet devices directly connected to the optical network drives relevant benefits in terms of CapEx, power consumption and occupied space in central offices. Furthermore, it enables a tight integration between packet and optical networks, which is of special interest as transport is dominated by Ethernet and IP traffic. For example, a single packet switch can provide both intra-data center (DC) traffic aggregation and, thanks to coherent pluggables, effective DC-to-DC interconnection. However, controlling packet-optical solutions requires a complete operating system that is much more complex than traditional NETCONF/YANG software agents employed in standalone transponders [Sgam-1, Gior].

SONiC (Software for Open Networking in the Cloud) is an open-source network operating system already deployed in production intra-DC networks and it is also considered a strong candidate to control packet-optical nodes although some operational extensions are needed to fill the existing architectural gaps. For example, SONiC does not natively support NETCONF and it does not encompass the needed software components to operate on coherent pluggable transceivers. Another gap to be filled is the coordination between packet and optical parameters on the same node, which are often provided by two different SDN controllers, one in charge of packet

resources and one in charge of optical transport. So far, this aspect is yet undiscussed in the scientific literature.

This section designs and implements a novel comprehensive workflow enabling coordinated control by SDN packet and optical controllers concurrently operating on a packet-optical node equipped with coherent pluggable modules and using SONiC enhanced with NETCONF/YANG components.

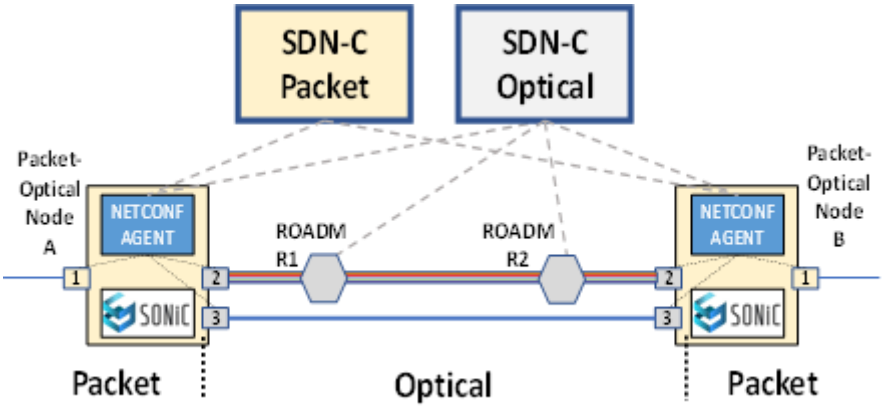

Figure 2.6: Disaggregated metro network scenario.

The reference disaggregated network scenario is illustrated in Figure 2.6. Two types of nodes are present: optical switches (ROADMs) providing optical switching and packet-optical nodes providing packet switching. Packet-optical nodes are equipped with pluggable transceivers. In large metro networks, a single controller with visibility on both packet and optical resources is hardly implementable due to scalability issues. Two controllers are then typically considered: an Optical SDN Controller (OptC) in charge of the disaggregated optical transport network and a Packet Controller (PckC) supporting Layer 2-7 configurations. Traditionally, each SDN controller has full visibility on all components and software modules of every controlled network element. However, in the considered scenario, two different controllers need to concurrently operate on packet-optical nodes. Thus, a proper workflow needs to be defined to enable the SDN agent of the packet-optical node to coordinate the operations imposed by each controller. Indeed, without proper coordination, complex multi-layer operations, such as recovery upon soft failure, would lead to management conflicts on the packet-optical nodes as well as to potential traffic disruptions.

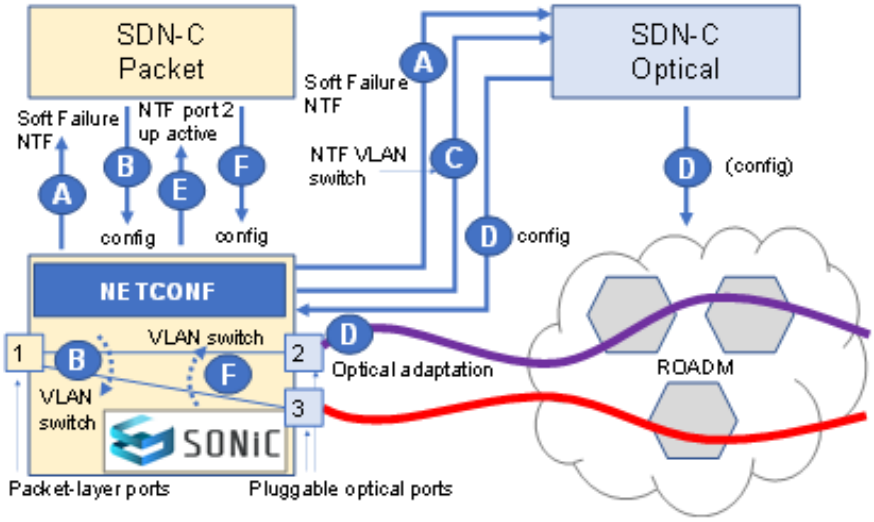

Figure 2.7: Proposed coordinated workflow for pluggable control.

The proposed workflow used to coordinate PckC and OptC operations is reported in Figure 2.7 (steps A-F). The workflow exploits the NETCONF-based SDN agent deployed in the packet-optical node. Both controllers are connected to the agent. To avoid non-standard and complex peer/hierarchical operations, the two controllers do not communicate each other to coordinate their actions. Instead, they leverage on the proposed workflow to avoid conflicts and guarantee segregation of control. Ownership segregation has been implemented exploiting the NCACM solution as detailed in RFC 8341. In particular, the OptC is provided with writing rights on the optical parameters and read-only rights (including enabling notifications upon subscriptions) on packet parameters. Similarly, PckC is provided with writing rights on packet parameters and read-only rights on optical parameters.

In the considered use case, two connections are configured on the network. Upon soft failure detection affecting the connection provisioned through the optical pluggable module (e.g., port 2 in Figure 2.6), a NETCONF notification is sent to both controllers (step A). This triggers PckC to initiate the recovery workflow, while OptC becomes aware of the soft fault but, to avoid concurrent operations potentially leading to traffic disruption, it does not enforce optical reconfigurations yet. In step B, PckC enforces new forwarding rules to an alternative pluggable module (from port 2 to port 3 in the figure) exploiting the protection connection (red in Figure 2.7). Then, step C triggers a further NETCONF notification to OptC, indicating that no tributary traffic is forwarded by the pluggable 2. This triggers step D: OptC can now enforce the optical transmission adaptation of the pluggable modules and the potential reconfigurations of the transit ROADMs. Once the adaptation procedure is concluded, the connection using pluggable 2 returns available, and a NETCONF notification is sent, notifying the end of the optical recovery (step E). This allows PckC to revert to the original state, successfully reconfiguring tributary traffic through the optical transceiver of port 2 (step F).

The proposed solution has been implemented in a network testbed reproducing the scenario of Figure 2.6. The packet-optical node architecture is depicted in Figure 2.8, it consists of a Mellanox SN2010 Ethernet switch running SONiC operating system over ONIE.

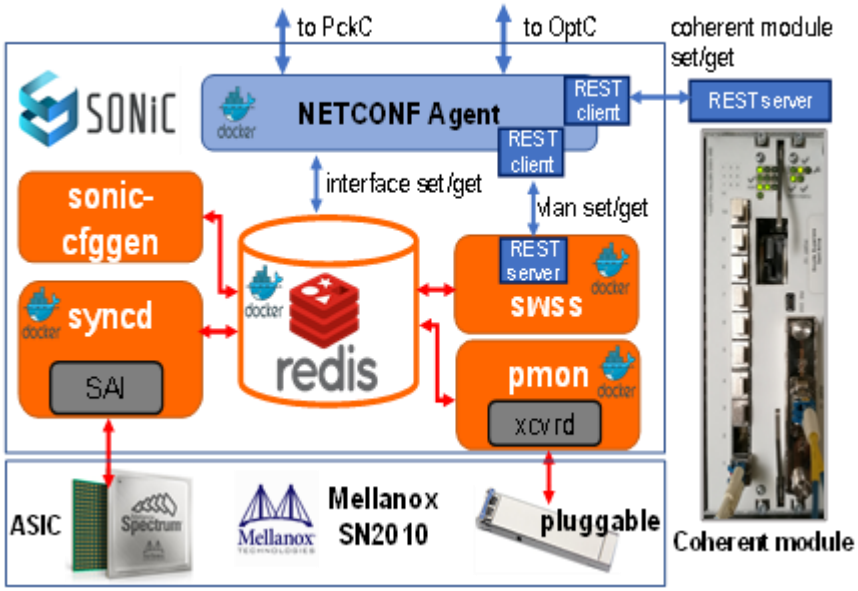

Figure 2.8: Packet-optical node based on Mellanox SN2010 and Sonic.

On top of SONiC, a specifically designed docker container runs the ConfD-based NETCONF agent communicating with the controllers. The container retrieves node status information by directly accessing the SONiC Redis database and enforces node

configurations using custom-built REST APIs integrated within SONiC. Node ports 1 and 3 are equipped with 10Gb/s SFP+ pluggable transceivers, monitored by SONiC pmon container; VLAN settings are applied through the SONiC swss and syncd containers. Port 2 is attached to an external 100 Gb/s coherent system configured as being a pluggable module, i.e., its driver is accessed via REST by the docker only, with no direct connection to the SDN controller as it would be for a standalone transponder. As also illustrated in Figure 2.6, Port 1 acts as tributary interfaces, port 2 handles the working 100Gb/s coherent communication across the optical transport network. Port 3 provides the alternative path for protection purposes. Under working condition, tributary traffic of port 1 is forwarded to port 2 only. Then, soft failure is generated by using a Variable Optical Attenuator (VOA).

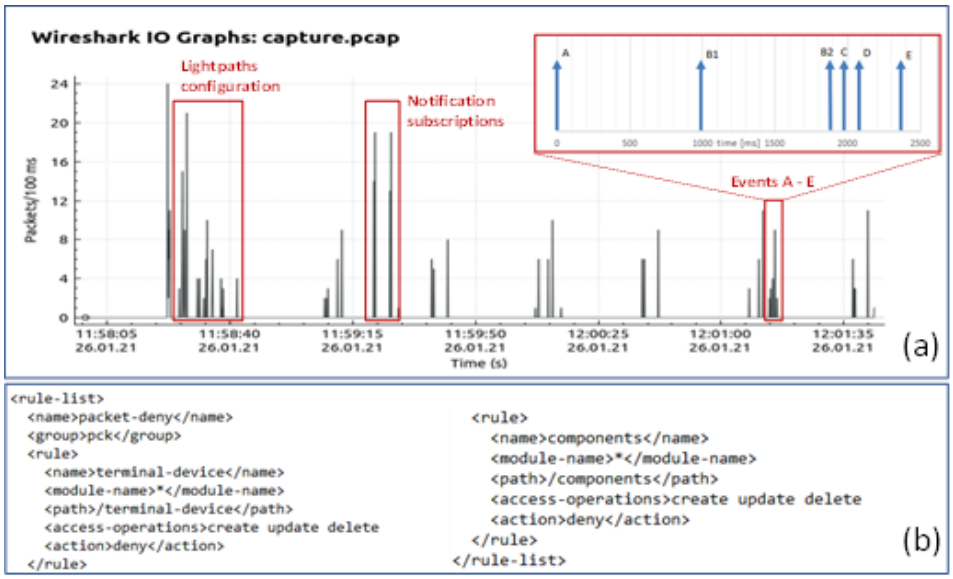

Figure 2.9: NETCONF messages captured between the agent and the controllers; xml scheme implementing RFC 8341.

Figure 2.9 shows the Wireshark capture summary of the NETCONF messages exchanged by the SONiC container and the two controllers. First, edit-config messages are exchanged to establish the two connections, second the two controllers subscribe to the notification stream of the agent, third the soft failure occurs and it is notified to the controllers that employ the proposed workflow. Other messages are periodically generated by the controllers for synchronization purpose. Workflow events are zoomed in the figure inset. In particular, failure notification is generated at time 0 (step A). Then PckC elaborates the rerouting options, e.g., exploiting the alternative path through port 3. After 996 ms (step B1), the container applies the received VLAN configuration thus rerouting the traffic on the backup connection (traffic rerouting is performed without loss of packets). The previous VLAN configuration is deleted at Step B2 (time: 1881 ms), triggering a new notification message (Step C time: 1984 ms) that informs the OptC that no traffic is anymore forwarded through port 2. Thus, OptC generates an edit-config message configuring a new operational mode (e.g., Forward Error Correction - FEC - adaptation) on port 2; this message is received at the agent at time 2072 ms (Step D). This configuration is applied at time 2372 ms (Step E), however the corresponding data plane operation requires around 38 seconds. During this time, port 2 becomes unavailable. Therefore, without the proposed coordination and traffic rerouting significant traffic loss would have been experienced, while, thanks to the coordinated workflow, no traffic disruption is experienced. The overall measured time between step A and E is always less than 2.5 seconds (out of ten experiments), with around 2 seconds required by the data plane configuration implemented in SONiC, while around 0.5 seconds are due to the control plane operations.

Thus, fast event coordination is achieved without experiencing traffic disruption, completing the workflow in less than 2.5 seconds mainly due to hardware configuration.

2.4. Hierarchical collaboration of SDN controllers

With respect to the previous section, this section explores the possibility to utilize a hierarchy of SDN controllers to operate on a packet/optical network including P4-based packet devices, hybrid packet/optical devices, and traditional optical devices.

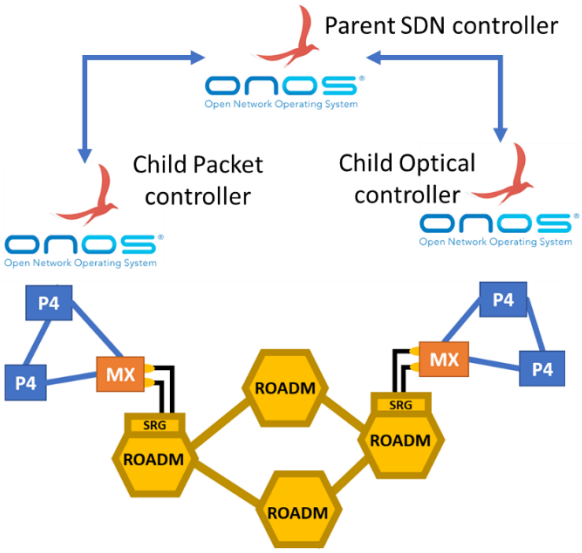

Figure 2.10: hierarchical controllers architecture.

The control of integrated packet-optical nodes requires the evolution of the currently available operating systems for packet nodes (e.g., Software for Open Networking in the Cloud - SONiC) to also support configuration, state information retrieving, and management of coherent pluggable modules. Indeed, following the traditional control plane architectures, packet-related configurations should be enforced by a an SDN Controller dedicated to the packet domain (i.e., PckC) while optical parameters need to be configured by another SDN controller dedicated to the optical domain (i.e., OptC). So far, the problem of coordinated control of packet-optical nodes by two SDN controllers has been addressed in [Ricc, Lope, Sgam-2], as detailed in previous section, relying on the Network Configuration Access Control Model (NCACM) solution detailed in RFC 8341. However, such solution may introduce significant maintenance problems especially in case of firmware and software updates at the node and at the controller level.

Specifically, this section shows an alternative approach based on inter-Controller communication. With this solution, packet-optical nodes only interact with the PckC. In turn, the PckC is enabled to configure optical parameters by proper interaction with the OptC, mediated by a hierarchical parent controller, as illustrated in Figure 2.10.

Figure 2.11 shows the proposed workflow to guarantee coordinated control of packet nodes, hybrid packet-optical nodes using pluggables, and optical nodes. The figure illustrates both the network initialization procedure (steps A-C), and the procedure used to activate a multi-layer connectivity service (steps 1-8).

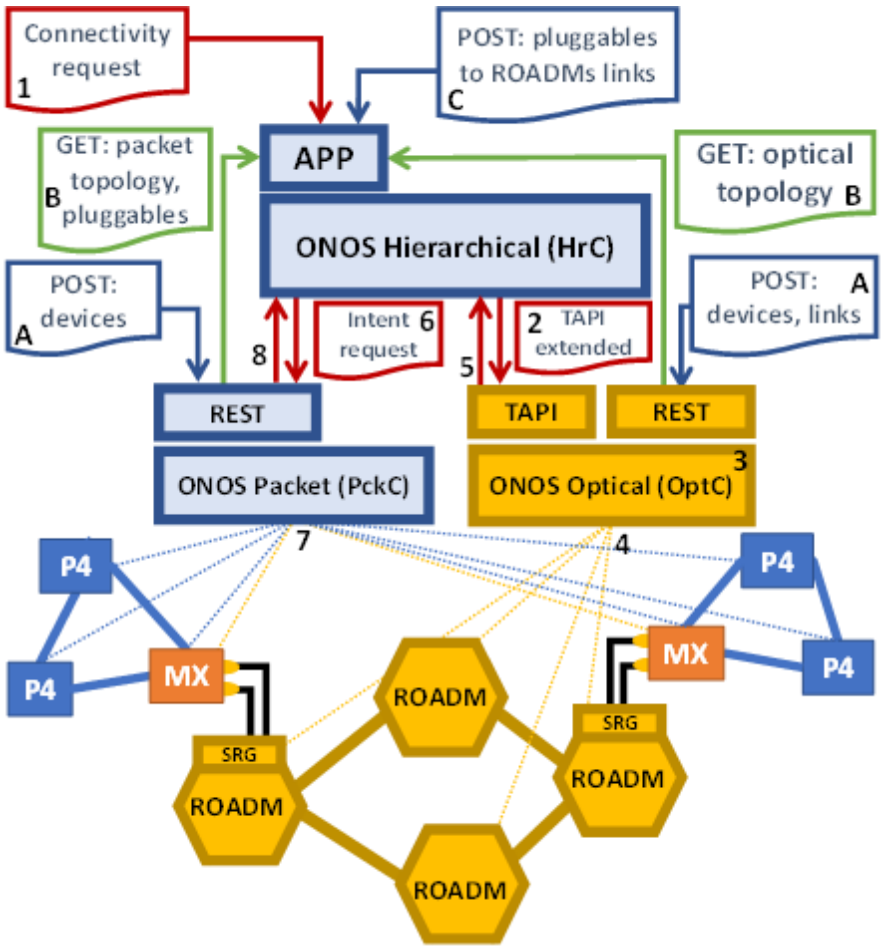

Figure 2.11 Control plane architecture and workflows. Letters A-B describe the network initialization workflow; numbers 1-9 describe the connectivity establishment workflow.

During network initialization the packet and the optical topologies are pushed into the respective controllers (step A); the hierarchical SDN controller (HrC) loads the topology of the two domains (including the pluggables modules discovered by the PckC) through the controllers REST APIs (step B); finally, the associations between the pluggables modules used in the packet-optical nodes and the ROADM add/drop interfaces are pushed into HrC (step C). All this data is classified as quasi-static information since determined by manual intervention and can be therefore initialized through specific configuration (i.e., POST commands on the REST APIs).

After network initialization, when a layer 2/3 connectivity request arrives at HrC (step 1), it first identifies the pair of pluggable modules to be interconnected through the optical transport network. At step 2, HrC sends a connectivity request (e.g., Open Transport API, T-API) between SRG connection points to the OptC. To effectively perform impairment-aware optical path computation, the OptC must be aware of pluggable supported features (e.g., supported modulation formats, FECs, operational modes). At step 3 the OptC performs impairment-aware path computation, identifying the suitable configuration for pluggable modules as well as traversed optical path. This step, typically, is not executed inside the SDN controller, but exploits external tools specifically developed with this target, e.g., GNPy [Mans, Ferr]. At step 4 the controller enforces the SRG-to-SRG configuration through NETCONF, driving the set-up of all traversed ROADMs. At step 5, once the path is successfully established, the OptC replies to the HrC informing about the available SRG-to-SRG connectivity as well as on the selected configuration of pluggable modules. Indeed, they cannot be directly configured since under the domain of control of the PckC. At step 6, the HrC generates a packet level REST connectivity request to the PckC. The request includes the configuration

previously identified by the OptC for the pluggable modules at the line side. At step 7, the PckC enforces the configuration to both involved packet-optical nodes, and other involved packet nodes. At step 8 the PckC informs the HrC about the successful configuration.

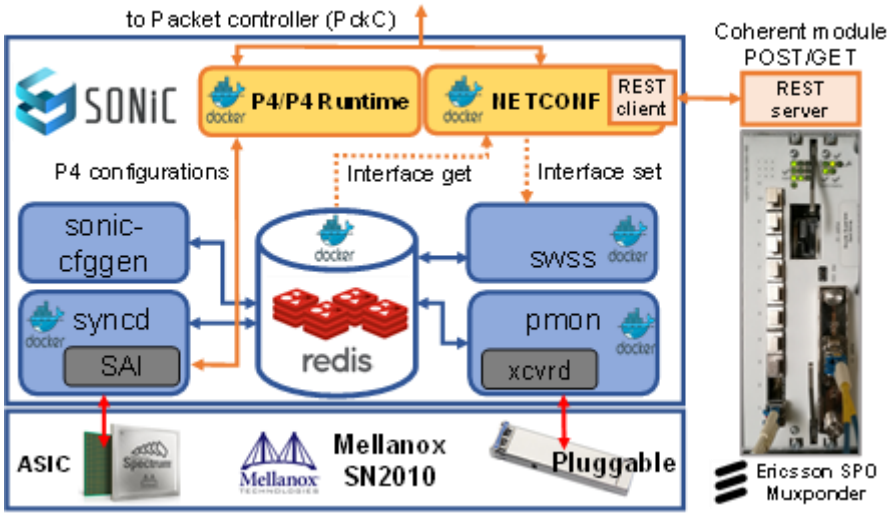

Figure 2.12 Hybrid-node architecture including P4-based and NETCONF agents, both connected to the Packet controller. Dashed interactions are implemented but not included in the demonstration.

The packet-optical node architecture is better detailed in Figure 2.12. It is an evolution of the architecture depicted in Figure 2.8, where (besides the NETCONF docker container to control the optical pluggables) SONiC also includes the a P4/P4Runtime docker container. Using these two containers, two parallel communication channels are established between the packet-optical device and the PckC to enable configuration of packet and optical resources, respectively.

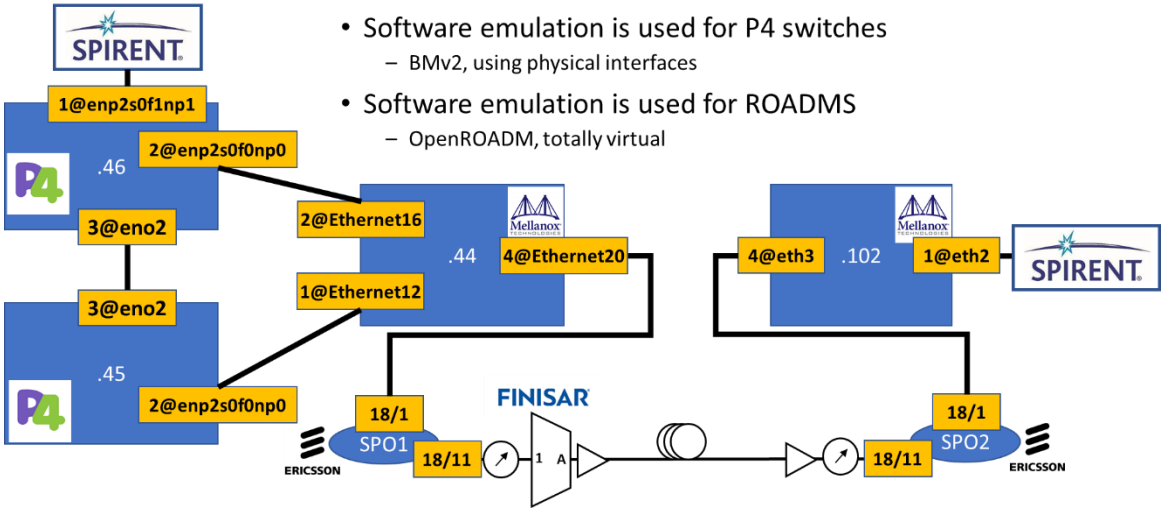

Figure 2.13: network testbed scheme

The illustrated setup has been demonstrated in a live experimental session in ECOC 2021 international conference. Specifically, Figure 2.13 illustrates the physical testbed that we have used including: two physical Mellanox switches adopting the internal architecture illustrates in Figure 2.12; two emulated P4-based switches adopting physical interfaces, two physical optical transponders, and four emulated optical switches (not represented in the figure).

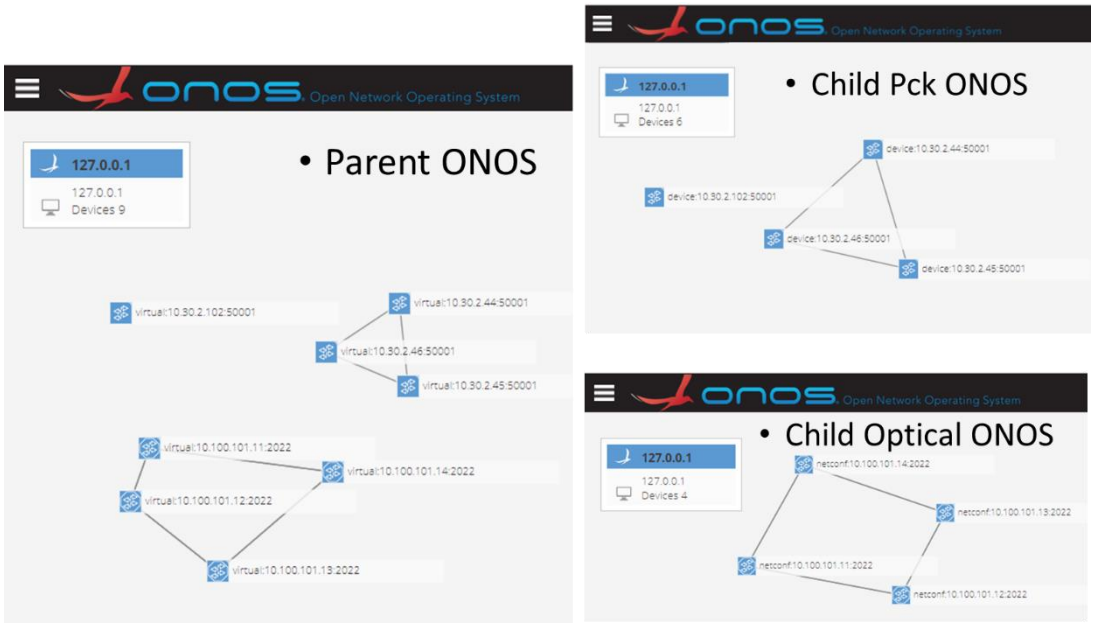

Figure 2.14: screenshots of ONOS controllers.

The live demonstration at ECOC demonstrated that the parent controller is able to correctly acquire the topologies of the child domains, moreover it is able to forward the required information from the OptC to the PckC required for the configuration of the optical pluggables installed in the hybrid packet/optical node. The whole communication required less than one seconds and is therefore suitable not only for the provisioning phase, but also for the management of failure recovery operations.

2.5. Investigating physical layer security (PLS) blockchain efficiency

2.5.1. Background

In WP2, we developed the concept of blockchain supported by two Guy Fawkes protocols, PLS and SLVP. The blockchain is a permissioned structure, which supports edge resource collaboration between participating IoT devices, and also between the Edge datacentres and the IoT swarm. These security mechanisms were analysed and developed under WP2 and it was demonstrated that they have the required security properties.

However, as we focus on low bitrate communications for IoT, we require the ability to address individual transactions in blockchain blocks without exchanging too much security data. LoRa communications for IoT are limited by EU regulations to a low duty cycle (1%; down to 0.1% in certain frequency bands). This severely limits the use of classical indexing structures, such as Merkle Trees or Merkle-Patricia Tries, especially when transactions on behalf of individual blockchain users are infrequent (which is the case with smart sensors, our target IoT category). Those indexing structures require a Merkle proof and its attendant data communications even when the target part of the block is missing (which would be in 90% of the time or even more) in the case of the PLS blockchain.

2.5.2. Context

Our lightweight blockchain protocol, SLVP requires the counterparties (IoT platforms) to satisfy themselves that their protocol messages are present or absent in every given

block, which would be costly if retrieval of such messages required active communication. The cost would be in terms of both energy/power and communication bandwidth. This motivated us to develop a novel solution for indexing individual contributions to a blockchain block.

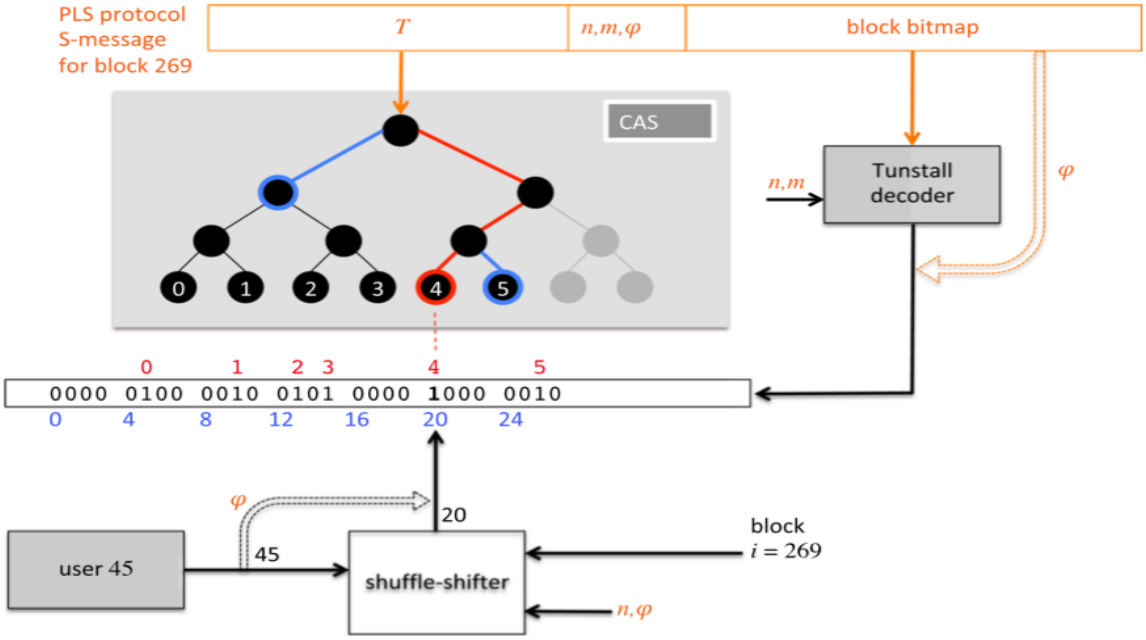

Figure 2.15: Indexing a block

We proposed and evaluated an indexing scheme based on a compressed version of the Merkle tree, which we called the Merkle-Tunstall Tree (MTT). An MTT consists of a dense binary tree possibly truncated on the right, whose leaves represent contributions from individual users to a given block. The space of user IDs is randomly permuted by our original shuffle-shifter to eliminate correlations between the presences of different IDs in a series of blocks. We call the result of permutation local IDs, meaning that they are local to the block. We associate with the block a bitmap where the absent local IDs are marked with 0s and the present ones with 1s. Due to the low duty cycle of IoT devices, most digits in the bitmap will be 0s for any given block. This points at low information content of the bitmap and the potential for efficient compression.

The Fog Server of the blockchain is the agent that forms blocks. It knows how many users contributed to a given block and, under the assumption that their presence is not correlated, it can apply a very effective compression technique. We use the so-called Tunstall compressor, which is based on this single parameter, namely the density of 1s. The block bitmap thus compressed is included in the block's Root of Trust together with the root hash of the dense, truncated Merkle Tree and some parameters; and the RoT is broadcast using the PLS protocol and architecture developed for BRAINE's WP2.

2.5.3. Investigation of efficiency

1. An IoT device checking a certain user's contribution (including of itself) to the current block will not need to communicate until it ascertains the presence of that contribution. It is done by examining the RoT, which the device receives for every block of the chain anyway. If the contribution is there, as in Figure 2.15, adjunct hashes of the tree need to be communicated along with the leaf to support the Merkle proof. The number of these hashes (the average length of the adjunct path) defines the communication volume. We investigated the adjunct path length by building a statistical theory of it and we found that 5 or 6 adjunct

hashes (making the packet size between 160 and 192 bytes plus the leaf size) are required, well within the LoRa packet size constraint (250 bytes).

2. We evaluated the effectiveness of our de-correlator, which is a combination of the modulo shift (adding a random number to an n-bit operand) and the perfect shuffle (rotation of n bits), which is shown as shuffle-shifter in Figure 2.15. We established that for the relevant n=1024, which is the expected maximum number of LoRa devices connected to a single hub, the number of rounds to achieve ~1% correlation between bits of the image under the standard avalanche test is 200. We observed that the implementation of a shuffle-shifter on a microcontroller has negligible cost (a few thousand instruction cycles).
3. The quality of the Tunstall compressor was evaluated for the relevant range of parameters. Table 2.9 quantifies residual redundancy of the compressor for the codeword size 4 and 8. We observe that even at w=4 (which only requires a tiny coding table) under 10% occupancy the redundancy of compression is below 8%. When the occupancy drops to only 5% the compressor does not perform as well, but then the entropy of the bitmap drops to 0.3 bit/digit and requires at least 300 bits (~37 bytes) to be represented. Whether it is 37 bytes or 37+30% ~ 50 bytes makes almost no difference for the size of the RoT. Any further reduction in occupancy can be accommodated by just listing the original user IDs, of which there would be less than 50.

Table 2.9: Quantifying residual redundancy of the compressor for the codeword size 4 and 8

w	p	κ	H_0	ρ(%)
4	0.05	0.37	0.29	30.4
4	0.10	0.50	0.47	7.5
4	0.15	0.65	0.61	6.9
8	0.05	0.32	0.29	13.4
8	0.10	0.49	0.47	4.6
8	0.15	0.63	0.61	3.8

Sample length before compression: 10^6. Column headers: w codeword length; p probability of 1; κ compression ratio; H_0 per-bit entropy; $\rho = (\kappa - H)/H$ residual redundancy(%)

3. Resource Management & Service Deployment

3.1. Resource Management & Service Description

3.1.1. Data model

The vocabulary for service deployment and resource management is based on Kubernetes schema as well as OWL-S through service profiles for resource and service description respectively. There is as well an additional metadata to enable the description of Docker Images, deployments and Services. Figure 3.1 gives an overview of the vocabulary developed for resource management and service deployment. Except for Docker Image and Service Profile which are Deployable subclasses, most of the model is a direct translation from Kubernetes Node schema. The vocabulary allows access to the current node status such as Allocable and the Capacity of the ComputationalResources, the internal and external network addresses as well as Node Information such as architecture. With this information, it is possible, for instance, to measure the capacity of the Nodes and devise more intelligent service allocations. The full vocabulary is available under Creative Common CC-BY-4.0 license at https://github.com/eccenca/braine-vocab.

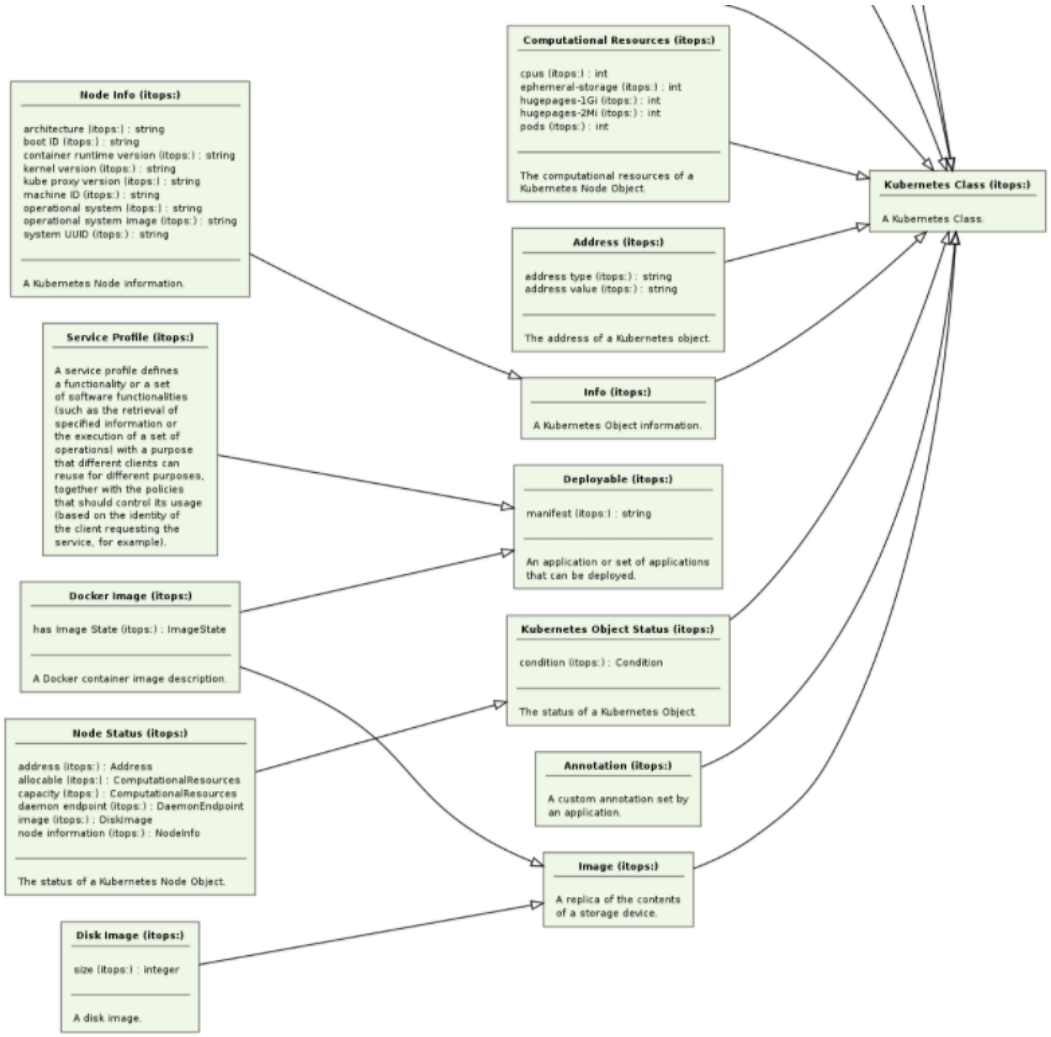

Figure 3.1: Excerpt of BRAINE vocabulary for Resource Management and Service Description.

There are discussions on going to decide how to parametrize Training and Test set collections for the AI models as well as frameworks and architectures. One way to go is to use the already existing OWL-S Service profile parameters, but the main question is how to parametrize the collection in a way that the running Cluster and POD has access while being restricted enough to not allow any other undesired or unplanned access. One way to go is to add an authentication mechanism as parameters such as user/password or public key. Another is to have a shared data space with restricted network access to BRAINE clusters, therefore the user could specify where to locate the AI collection in the shared private data space. Following this idea, one could also upload the collection metadata such as location and size to eccenca Corporate Memory, allowing users to easily check, query and select the desired one.

Registry Interfaces: In addition to the vocabulary, we have also developed the interface to allow partners to register their applications and services, creating the service catalog for deployment. Figure 3.4 displays the Docker Image register window in CMEM, it allows users to register Docker images for deployments. Figure 3.5 displays the Service Profile Register Window that allows the registering of Services through Kubernetes Deployment description files. In both windows there is an attribute *manifest* which is used to either register Kubernetes Deployment descriptor (Figure 3.2) in case of Service Profile and Docker Image Descriptor (Figure 3.3) in case of Docker Images.

```
kind: Pod
    metadata:
      labels:
        run: helloworld
      name: helloworld
    spec:
      runtimeClassName: rune
      containers:
      - command:
        - /bin/hello_world
        env:
        - name: RUNE_CARRIER
          value: occlum
        image: helloworld
        imagePullPolicy: IfNotPresent
        name: helloworld
        workingDir: /run/rune
    EOF
```

Figure 3.2: Kubernetes manifest example.

```
FROM alpine

CMD ["echo", "Hello BRAINE!"]
```

Figure 3.3: Image manifest example.

The Docker Images have an additional attribute called state. The state is used to indicate whenever an Image is *New* and therefore needs to be reviewed, if it is *Under Review,* or if it is *Ready* for deployment.

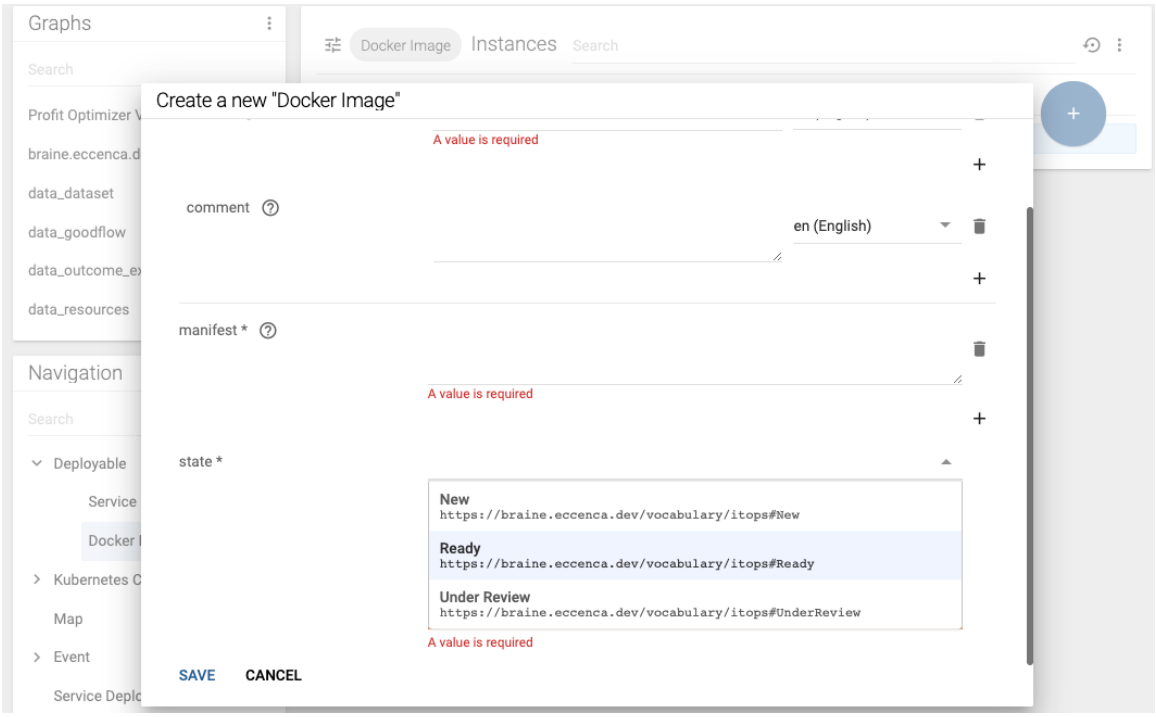

Figure 3.4: Docker Image Registry Window.

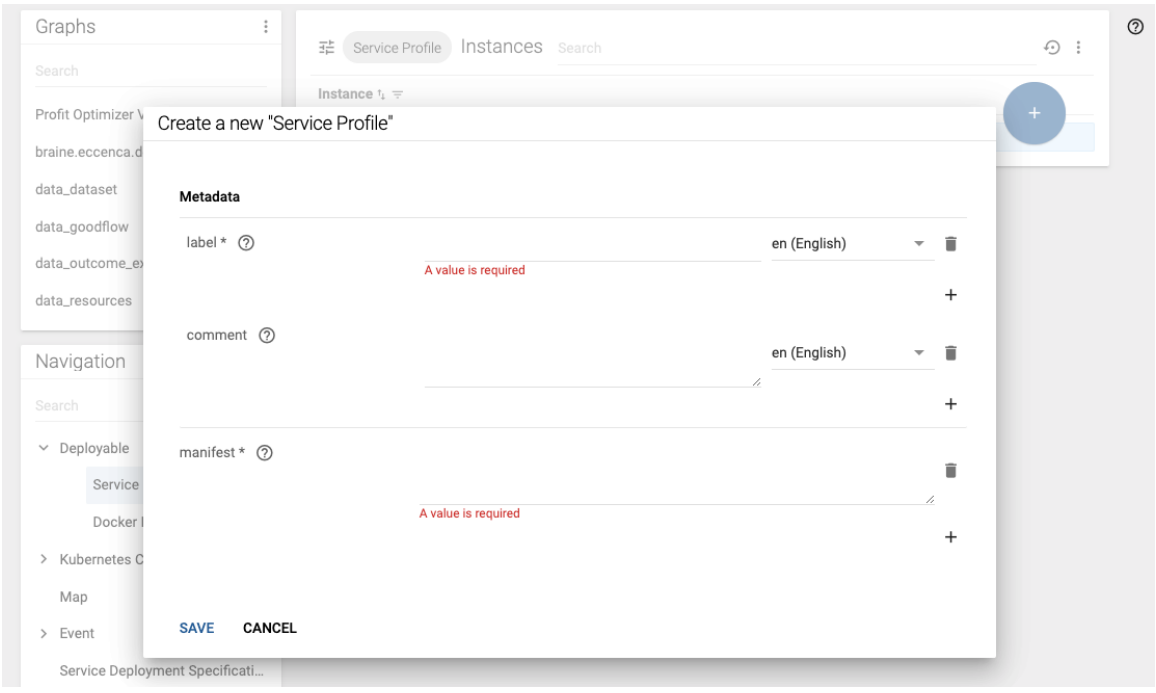

Figure 3.5 Service Profile Registry Window.

Onboarding: The service registration is kickstarted by an onboarding process where all users are asked first to create a docker image of their service that will be later placed in a Kubernetes Node. The image is registered through the process depicted on Figure 3.6 as follows. The user registers the Docker Images through the dialog on Figure 3.4**.** The inserted image is marked as new, and stored in Corporate Memory Platform. The cluster admin verifies if the image was properly created, taking into account privacy as well as access issues and marks it as an *Under Review*. When the cluster admin certifies that everything is correct, the image then receives the state *Ready*, meaning the Image is ready to be used.

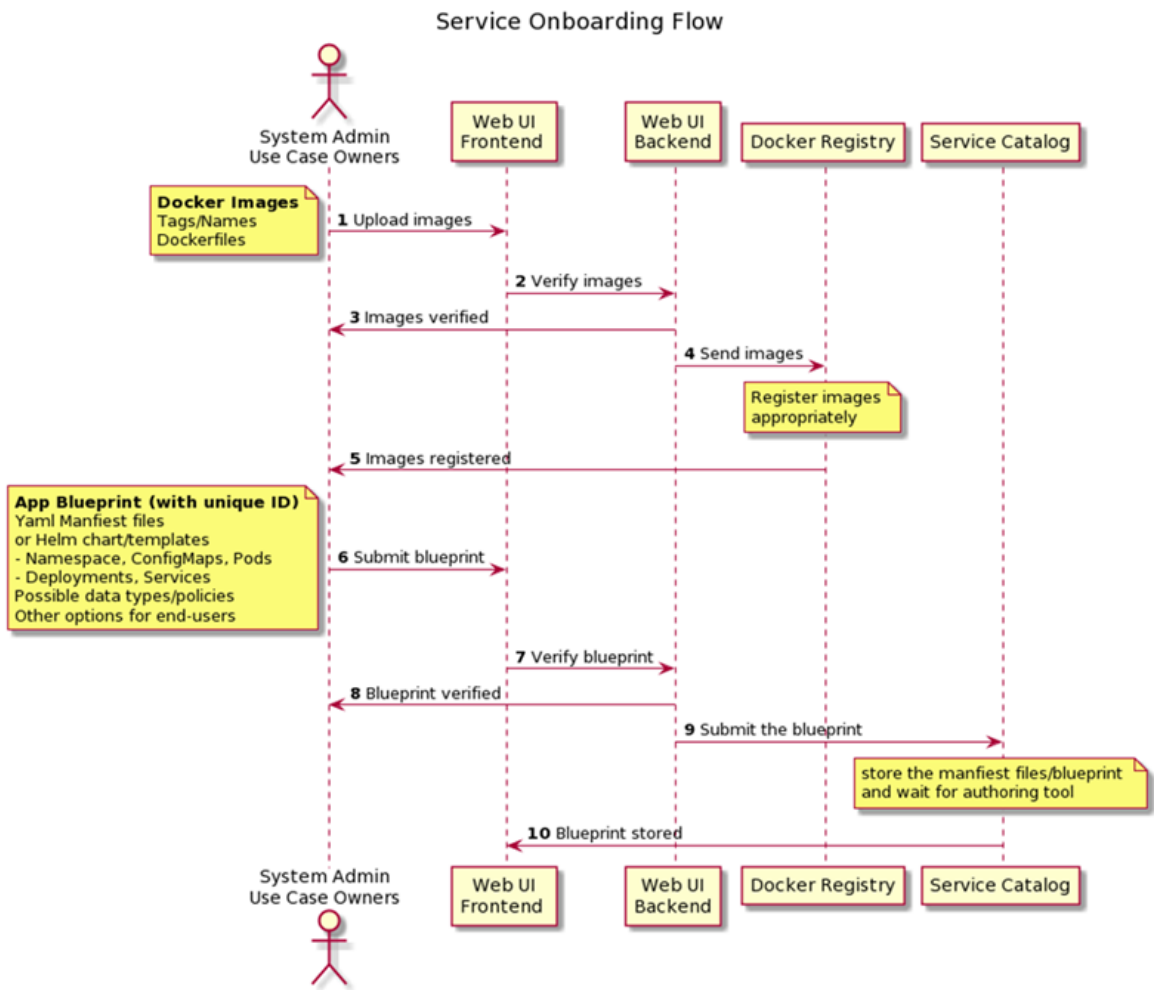

Figure 3.6: Service Onboarding Flow.

3.1.2. Resource & Service Orchestration

The Service & Resource Catalog instantiation relies on different components, each of the necessary for an operational system functioning. The Metrics Relay is an application that extracts information from Kubernetes APIs such as ComputationalResource allocation and capacity from the Nodes (see T3.3, resources available as well as their condition. The Relay also makes use of the Kubernetes Metrics server that provides periodic updates on Memory and CPU consumption. The relay makes use of the eccenca Corporate Memory Data Integration module to perform transformations on the Kubernetes data extracted from the APIs and populate the BRANE Knowledge Base. The relay is open accessible at https://github.com/eccenca/braine/tree/main/relay (see Figure 3.7).

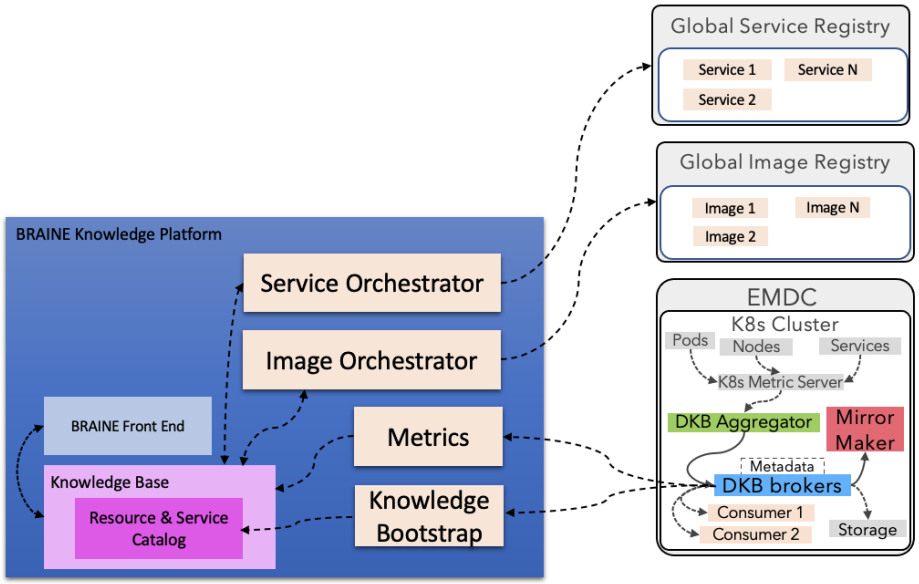

Figure 3.7: Service & Resource Repository components.

Decoupled Information Transferring: To facilitate the transferring of information between the Kubernetes Cluster (Node) and the CMEM platform a bootstrap system was developed. The bootstrap system reads information from the Kubernetes Cluster or from DKB broker in a specific address and pushes it to the eccenca Corporate Memory using a transformation from the CMEM Data Integration module (available at https://github.com/eccenca/braine/tree/main/setup). CMEM is a semantic data management software that accelerates analytics and reporting projects by transforming the way enterprises understand, align, prepare, and access their data. In the BRAINE project, CMEM is used to store and manage the BRAINE knowledge graph that contains information about Kubernetes Clusters, Nodes and Pods as well as services and workflows transformations through mapping rules. The BRAINE project uses the Data Integration module (Figure 3.8) to ingest Kubernetes information such as Nodes and their resources (memory and CPU) to populate the BRAINE Knowledge graph. The Corporate Memory instance used in BRAINE project is accessible at http://braine.eccenca.dev.

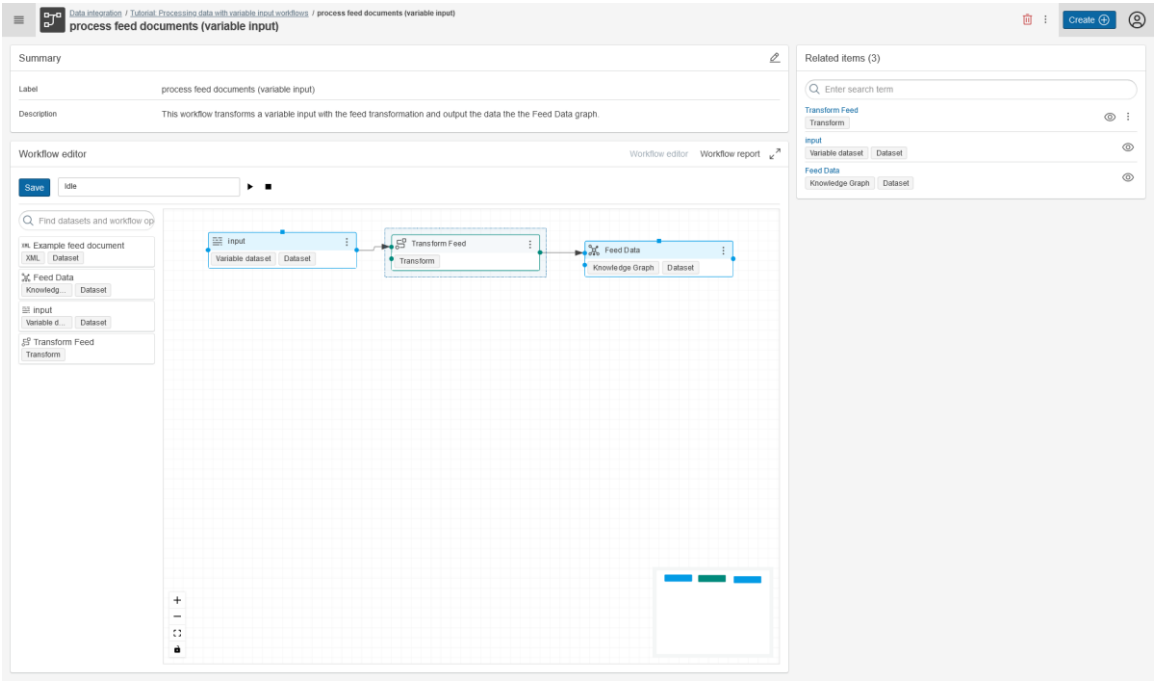

Figure 3.8: Corporate Memory Data Integration module.

CMEM contains as well a customized version of Redash (https://redash.io/) that allow users to create personalized dashboards using SPARQL queries. In the context of BRAINE project, the Corporate Memory Redash (Figure 3.9) is used for monitoring Kubernetes Nodes CPU and memory consumption.

To orchestrate changes among the physical and the semantic abstract objects in the repository. The image & service orchestrators were developed. The image orchestrator is designed to register, update or remove images at the Global Image Register (GIR). When an image needs to be registered, the Image Orchestrator reads the image information from the BRAINE image registry and register it in the GIR, updating its status to ACTIVE (see https://github.com/eccenca/braine/tree/main/container-orchestrator). The registered images will then be available to build Service Descriptors and ultimately perform service deployments.

The Service Orchestrator is responsible for service deployments, collecting service metadata and maintaining the information of the Deployments at the BRAINE Resource & Service Repository synchronized with the information of the running service. It supports Service Deployment and Partial Status synchronization features. When a user defines a Deployment at the Authoring tool a new Deployment is instantiated at the BKB, the Service Orchestrator recognizes the new service through the state and reads its Service Deployment Specification containing information such as the Docker deployment, the user constraints, and the target Kubernetes Cluster (Node). Notice that a deployment may or not contain a Kubernetes Node or constraints as those attributes are not mandatory. In case a Node is not specified but there are constraints, it then verifies which Node is suitable for running the service, checking the resources of the Nodes available at the BKB. If no suitable Node is found, the Deployment will be in waiting state until a Node containing the constraints is found. In case the Service Deployment Specification has no constraints, the Service Orchestrator will use the first Node available. Otherwise, it will use the Node specified. When a Deployment is successful, the Service Orchestrator changes the Deployment state for running or stops with error otherwise.

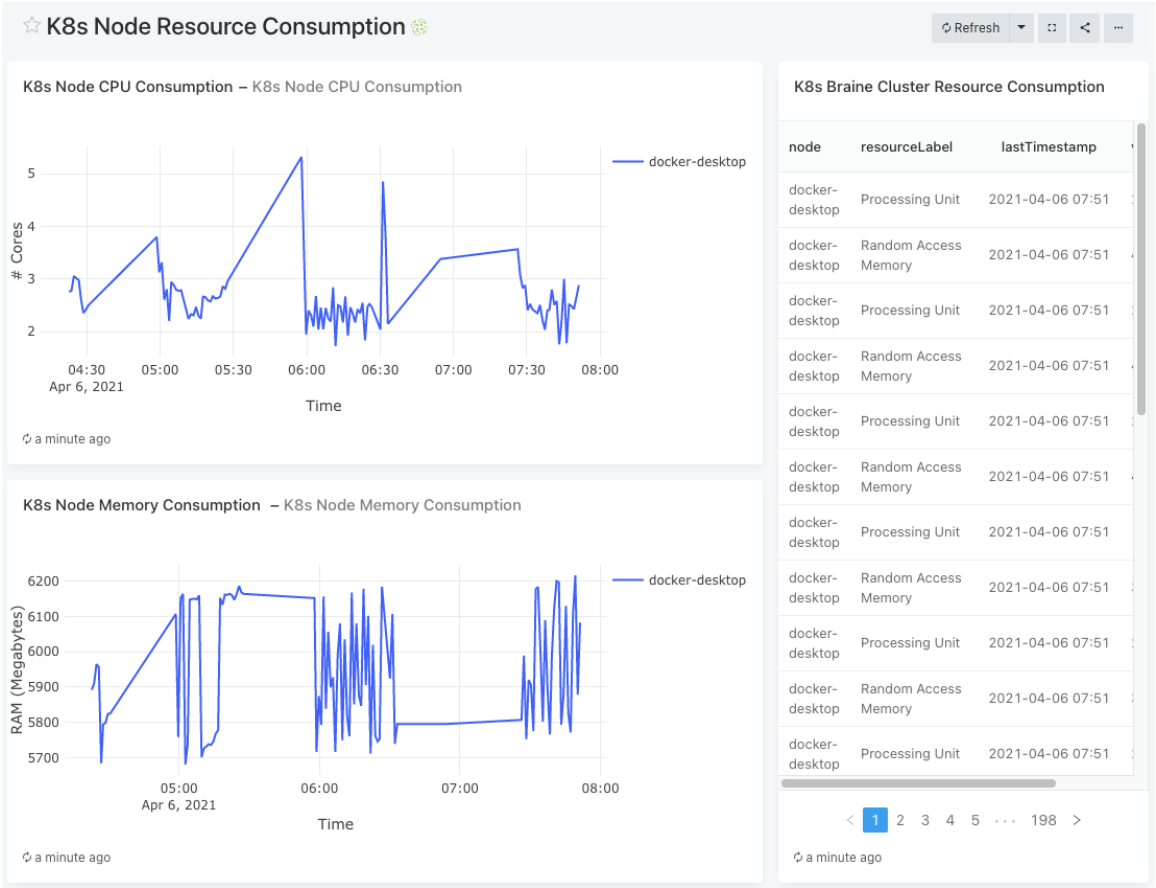

Figure 3.9: Corporate Memory Redash module showing Node's CPU and Memory consumption.

3.1.3. Semantic Web

The service offering is to address distributed and heterogeneous systems, to provide unified and centralized resource APIs to the workload distribution and service management.

Semantic Web is needed to deploy this service and understand the complexity.

The qualified data has to be understood before using it.

During this process, it has to be clarified where the data is coming from and where the information should be used afterward.

The right data has to be used and executed at the right time.

3.2. MEC platform applications deployment

In D3.1 we have investigated the MEC platform architecture, based on Intel OpenNESS framework. Such opensource platform has been customized for Braine application to meet the objectives defined for Edge computing. In this Deliverable, we will focus on the applications deployment which is an essential step that has to be done to onboard the services into the MEC platform.
Each tenant application has to be virtualized (Docker containers or VMs) and uploaded in the Braine registry that resides in the MEC Controller. Once the application has been uploaded, from the MEC controller it is possible to deploy a service by using a GUI

interface. The deployment phase is in charge of downloading the tenant MEC app image to the Edge node and, then, starting the service. In the following, this procedure is summarized.

3.2.1. Creating application

Before uploading the application image to the Braine registry, the service to be deployed has to be defined in the MEC controller platform. Adding the application to the MEC controller is straightforward thanks to the GUI. Figure 3.10 shows how to add the service from the GUI. Special attention should be paid to the fields required to describe the service, also in terms of computation capabilities. Here is the list of the fields filled with an example for a better comprehension:

- Name: TestApp
- Type: Container
- Version: 1.0
- Vendor: Sma-RTy
- Description: application test to be onboarded
- Cores: 2
- Memory: 1024
- Source: https://braineregistry/smarty_app/smartyapp.tar.gz

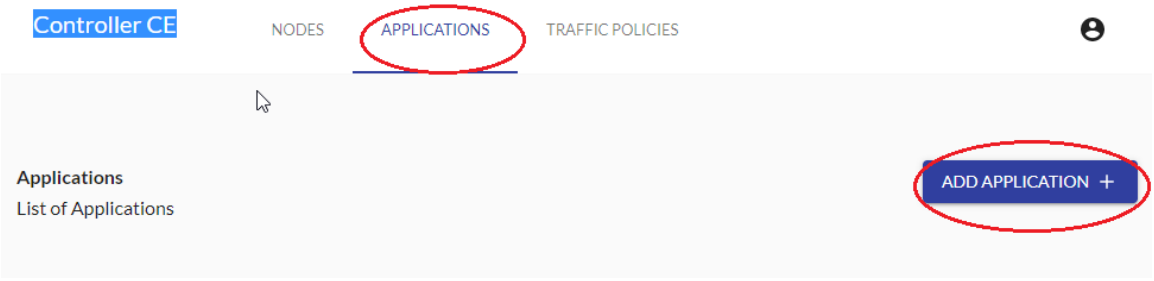

Figure 3.10: Creating application in the MEC Controller

3.2.2. Deploying application

Once the service has been registered, we are ready to deploy the application. From the main window of the Controller GUI, you can see the list of the Edge nodes available. For each Edge node you can define the MEC apps (available from the list of applications described in Section 3.2.1), the network policies (if any) and you can check the status of the network interfaces. By clicking on “deploy app”, you can select the service you would deploy and then the system starts to download the image to the edge node from the Braine registry. Figure 3.11 depicts this step.

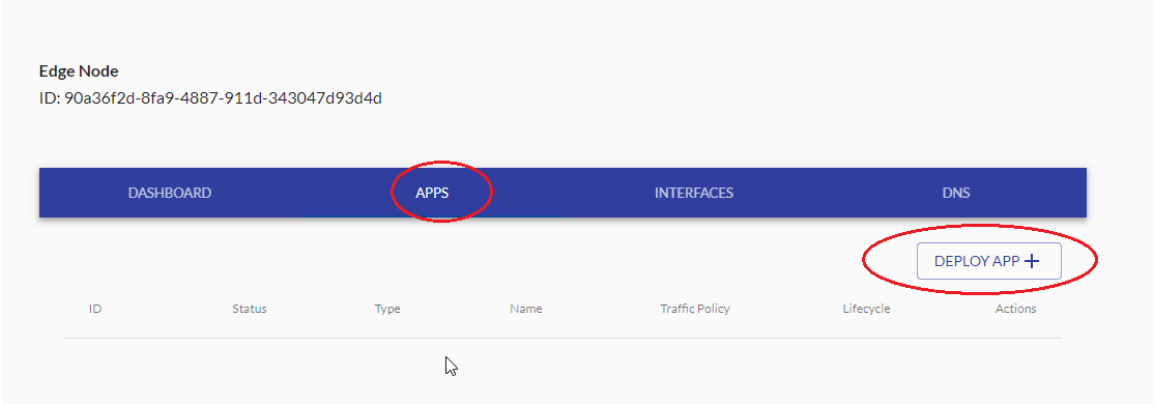

Figure 3.11: Deploying an application to the Edge node

3.2.3. Start/Stop service

After downloading the service image to the Edge node, we are allowed to start, stop, restart and delete the application. By clicking start, the service starts in the Edge node and you can directly control the application log from both Edge controller and Edge node. Indeed, all the system logs are sent to the controller which is also in charge of monitoring the application status. Figure 3.12 summarizes this last step.

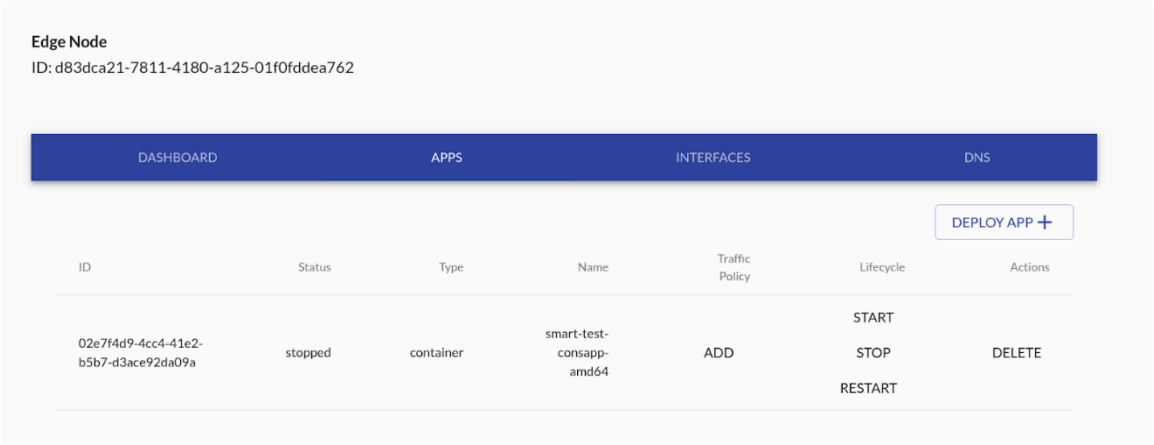

Figure 3.12: Start the service in the Edge node

3.3. SLA broker in distributed edge environment

The BRAINE SLA Broker's role is to notify the orchestration framework about any violation of the SLA agreement. The BRAINE framework implemented two-level SLA management levels: local and global, to reach an efficient SLA violation triggering system in a distributed edge environment. The local level consists of a data collector, analyzer, and local policy manager. The data collector gathers and aggregates the measurement points received from the telemetry system. The measurement points are aggregated based on predefined factors related to the subject application (e.g., every second or 10s). The analyzer is responsible for informing the local policy manager in case of SLA violations. The local SLA manager performs an action based on the SLA agreement. For instance, as in Figure 3.13, the SLA Broker could inform the local orchestrator regarding the SLA violation, so the local orchestrator could have rescheduled the instance in another host belonging to that specific edge node. In case of not resolving the SLA violation for a specific time/iteration, the local policy manager escalates the issue to the global policy manager, which requits the global service

orchestrator to resolve the issue (i.e., by rescheduling the instance on another edge node).

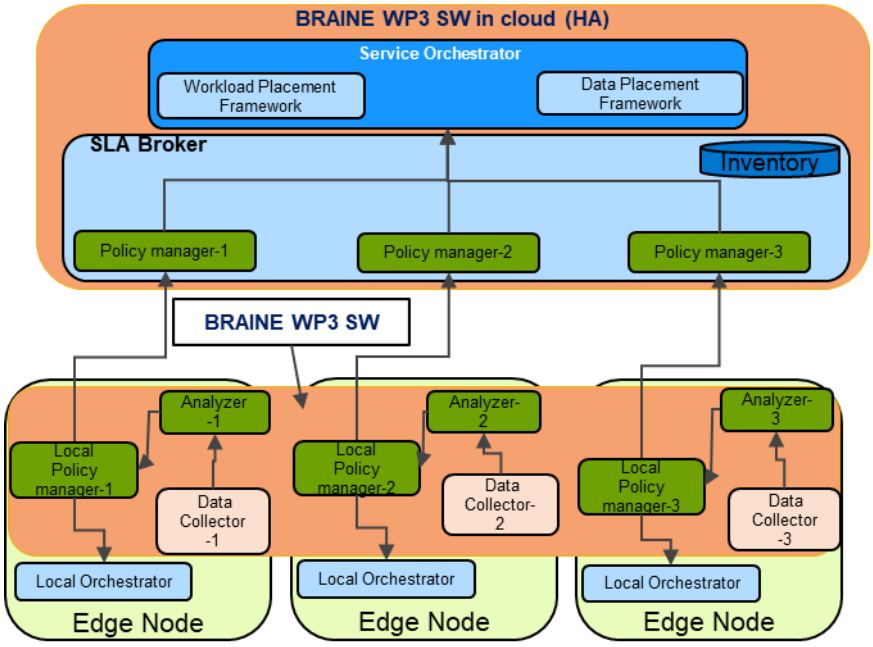

Figure 3.13: Distributed SLA Broker Architecture.

4. AI/ML-based workload placement

4.1. AI/ML-based scheduler

The BRAINE scheduler (available at: https://gitlab.com/braine/wp3-work_placement-luh/) customizes the default behaviour of the Kubernetes scheduler by using deep reinforcement learning (DRL) in the node scoring step. More specifically, it uses several interactions with the EMDC environment to learn an optimal node scoring strategy. This will result in node selections that optimize a long-term objective, such as maximizing the resource efficiency in the cluster and as a result its energy efficiency. To do so, it uses the following information in the RL state:

- Pod features: The CPU, memory and disk requests of the pod.
- Node features: The current resource utilization levels of the nodes across the 3 resource dimensions (CPU, memory, disk).

This information is then fed into a neural network that is trained to return the node scores. The reward/objective to be optimized can be specified in the configuration file prior to the training process. The optimization options that are currently available are:

- OP1: Minimize the number of active nodes, while minimizing the performance degradation that arises from placing multiple pods on a node experiencing resource contention.
- OP2: Minimize the long-term average wait time of workloads.

The training process is based on the Double Deep Q-Network (DDQN) algorithm with a Prioritized Experience Replay (PER) buffer. PER prioritizes the most useful experience tuples in the training process, instead of selecting them completely at random. Other algorithms allowing to deal with a varying number of worker nodes are currently being investigated.

As mentioned before, the BRAINE scheduler customizes the scoring step. Scoring is the step where all of the feasible nodes (obtained from previous steps of the scheduler) are ranked so that the node with the highest score will be selected to host the pod. A high-level illustration of the different components involved in the proposed RL-based scoring plugin is presented in Figure 4.1.

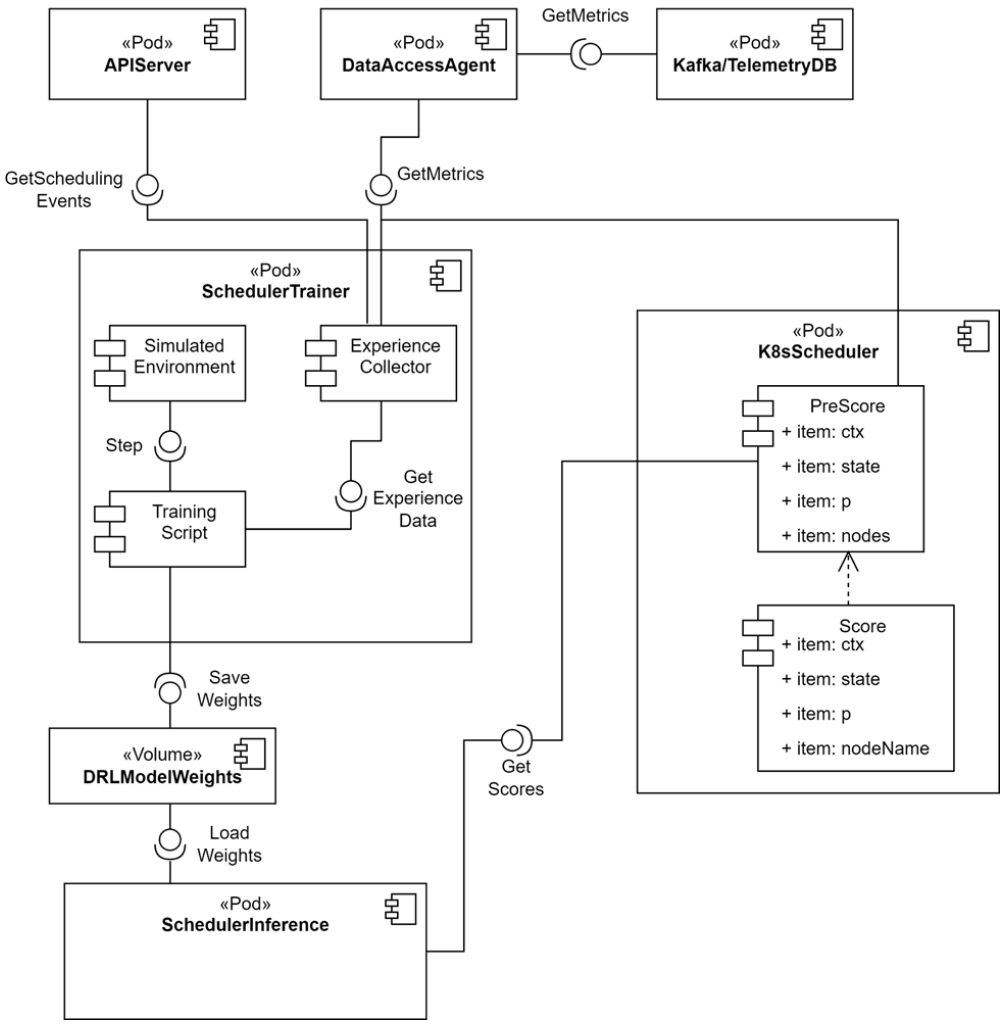

Figure 4.1: Component diagram of BRAINE RL scheduler

1. **Scheduler Trainer**: is the training component that is deployed as a pod and is in-charge of training the neural network for various cluster sizes, training data, and optimization objectives. Currently, the neural network is trained based on interactions with a simulated EMDC environment. However, later on, training will be performed based on real cluster data collected by the ExperienceCollector in order to continuously adapt to the real workload patterns. Trained models including network structure and weights are persisted (in a volume called DRLModelWeights) and delivered to SchedulerInference for the serving phase.
2. **Scheduler Inference**: is a containerized Kubernetes service hosting the ML-based inference engine. The inference engine serves the prediction/scoring requests based on the trained models produced and deployed by SchedulerTrainer. The inference engine functions are exposed via a RESTful API.
3. **BRAINE K8s Scheduler**: is the Kubernetes scheduler that its scoring plugin has been replaced by the LUH developed custom scoring module. This component also runs as a standalone pod. Its scoring plugin implements the interfaces of the PreScore and Score extension points. In particular, the PreScore function interacts with SchedulerInference to retrieve the node scores based on the current cluster and workload states. The state is formed by pod features and nodes' features. The required pod features are its resource requests which can be retrieved from the *p* parameter of the PreScore function. As for the nodes' features, they correspond to their current resource utilization levels, which are

retrieved from the Data Access Agent pod. The data access agent pod is indeed a component of the cognitive framework (see next sub-section) and can be used by multiple ML-based components, systems, and partners. The obtained node scores, from the inference engine, are written into the PreScore state variable. Since the Score function also has access to this information, it retrieves the scores and uses them for the evaluation of the highest-rank node.

4. **Data Access Agent**: is a standalone containerized Kubernetes service that as a component of the cognitive framework exposes a REST API and acts as an intermediary between the scheduler and the telemetry data provider or any other data source of interest for the AI/ML modules. At the moment cluster state is chosen to be stored and acquired from the telemetry database, hence the Data Access Agent retrieves nodes' features from there. However, other components/ partners can replace the data access logic of this component to acquire data from other sources

Table 4.1 outlines the different interfaces provided/required by the aforementioned components.

Table 4.1: Specification of the Interfaces of the components of the BRAINE scheduler

Interface name	Provided by	Required by	Input data	Returned data
Step	SimulatedEnvironment	TrainingScript	Selected action (machine)	Updated state, reward, a flag indicating the end of the episode
GetSchedulingEvents	APIServer	ExperienceCollector	Period of interest for getting the scheduling events.	Scheduling events of pods. More specifically, scheduling time, pod info, assigned node.
GetMetrics	DataAccessAgent	ExperienceCollector	Period of interest Metric of interest (in this case, the node util zation level for each resource dimension)	Node utilization metrics at the time where those scheduling events occurred.
GetExperienceData	ExperienceCollector	TrainingScript	-	Experience tuples in the proper format.
SaveWeights	DRLModelWeights	TrainingScript	Path where the weights of the trained NN are saved.	-

LoadWeights	DRLModelWeights	SchedulerInference	Path where the weights of the trained NN are saved.	-
GetMetrics	DataAccessAgent	PreScore (K8sScheduler)	Name of the metric of interest. In this case, real-time utilization rates of the nodes across all resource dimensions.	Values corresponding to the requested metrics.
GetMetrics	TelemetryDB	DataAccessAgent	Name of the metric of interest. In this case, real-time utilization rates of the nodes across all resource dimensions.	Values corresponding to the requested metrics.
GetQValues	SchedulerInference	PreScore	Pod features (requests in terms of CPU, memory, disk) Nodes' features corresponding to their utilization rates across all resource dimensions.	List of Q-values corresponding to the current RL state.

4.2. Cognitive Framework

During the project, many partners realized that different AI/ML software elements are under development. These elements share many characteristics, including their need for training and serving, their need to persist models and weights, and their need for cluster and application-level telemetry data. In order to better address this class of requirements a cognitive framework was proposed and is gradually getting integrated into the system. Figure 4.2 illustrates the initial architecture of such a framework.

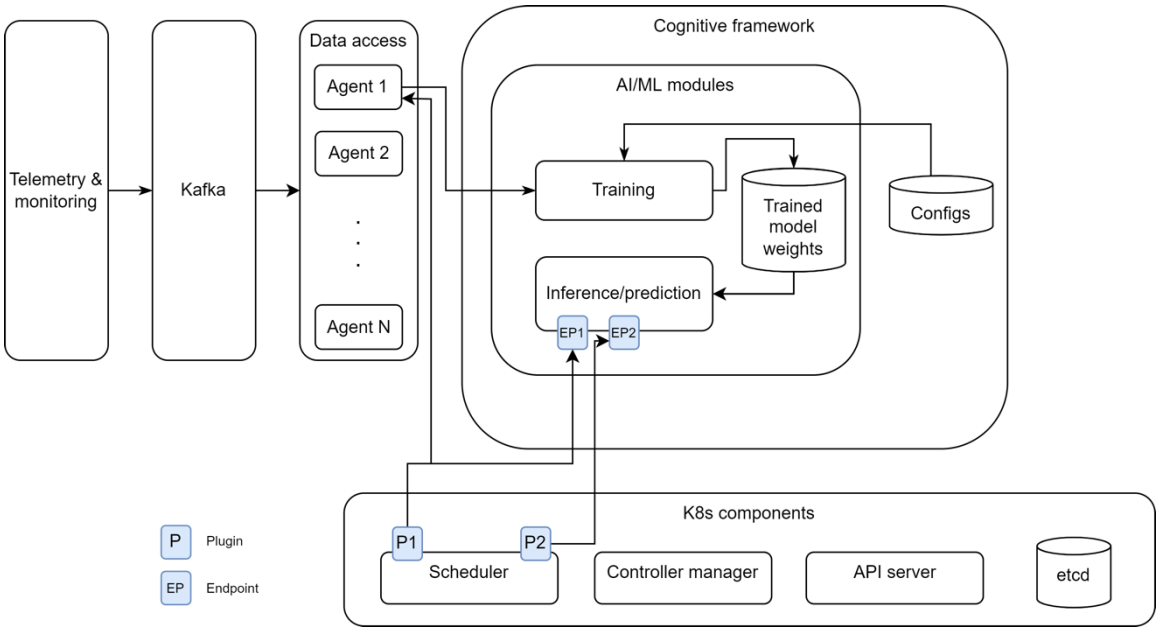

Figure 4.2: The Architecture of the BRAINE cognitive framework

Each partner registers its training and serving modules with the framework and provides access to the serving agents (the inference/prediction components) via endpoints (preferably RESTful). These endpoints will be consumed by plugins or other components e.g., the scheduler in Figure 4.2. On the other hand, any training module that needs accessing any data source is required to perform the action via the data access agent. The serving modules are recommended to not directly acquire data from external data sources neither directly nor via the data access agent(s). They should rely on the API calls to their endpoints and collect all the required data via parameters passed to their interfaces. Similarly, they should return their predictions as responses to the API calls. Continuing on the recommendation to implement RESTful APIs, the data exchange between the consumers and the APIs is recommended to be JSON.

Each partner working on AI/ML-based workload placement will be responsible for implementing their data access agents to obtain data from the telemetry system, the message queues, distributed knowledge base, or any other data source. Later on, the developed data access agents can be merged to have a more generic system for obtaining data. That can be plugin-based or a descriptive YAML-based data acquisition system that different partners can utilize and configure towards their needs.

4.3. Workload prediction and placement of vRAN

We consider a system architecture in Figure 4.3 where the L2 layer (MAC, RLC) and L3 layer (PDCP, RRC, and SDAP) layer is considered as virtualized network function (VNF) and is deployed at the edge micro data centre (EMDC). The architecture which is shown in the Figure 4.3 aims for dynamic adaptation of underlying infrastructure of 5G radio units i.e., the RRHs. At the first step, the metrics which will be used for the prediction algorithm are collected from the 5G open-RAN radio stack gNB network function and they are delivered using data bus of EMDC to the Resource Manager (RM) entity. A predictive technique is defined as a statistical model that can be applied to known data of a given phenomenon to estimate future information. The RM utilizes this data to feed arbitrary prediction techniques based on Markov decision process and/or sequence model in the form of MDP available at the prediction block, with proper inputs that allow

characterization of the virtualized gNB operation regarding its demand for computing resources of the EMDC.

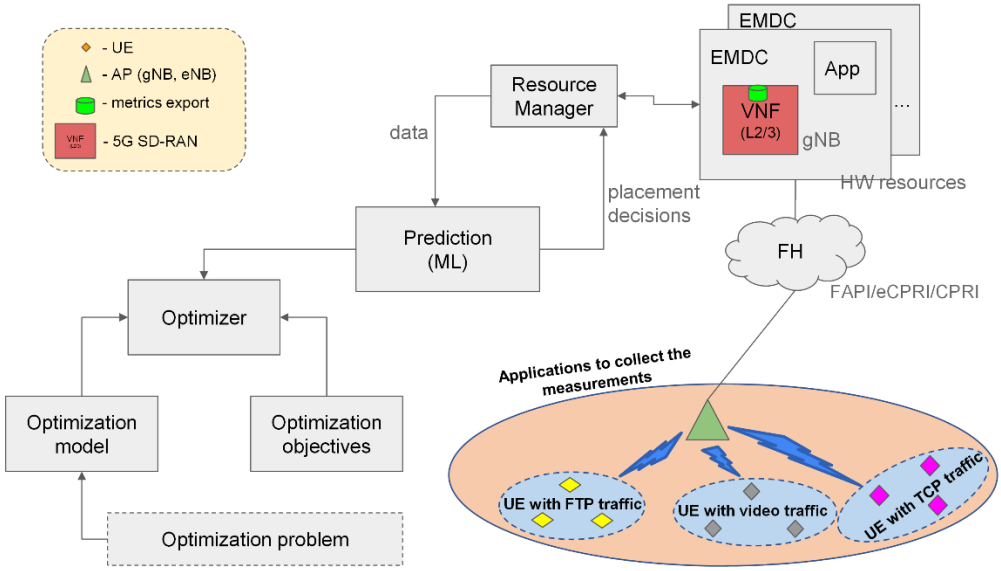

Figure 4.3: The architecture of workload forecasting and prediction.

With the EMDC design of edge datacenter, connected via optical infrastructure that is dynamically managed by SDN, fine grained capabilities for VNF/CNF placement are available. The architecture that is proposed in the Figure 4.3 aims at dynamic adaptation of underlying infrastructure of 5G radio units i.e., the remote radio heads (RRHs). The objective here is to activate/deactivate in the temporal domain certain frequency subcarriers and/or particular RRH to be used by certain UEs based on forecasting the workloads of the gNB. The workload forecasting steps are described as follows: firstly, the metrics are collected from the 5G open-RAN radio stack gNB network function and they are delivered using data bus of EMDC to the Resource Manager (RM) entity. The RM utilizes this data to feed arbitrary prediction algorithm (mainly based on model-free approaches) available inside of the prediction block, with proper inputs that allow characterization of the virtualized gNB operation regarding its demand for computing resources of the EMDC. The prediction techniques to forecast the vRAN as a workload we consider the signal-to-noise ratio (SNR) metric from the gNB protocol stack by running several applications at the gNB and UE sides during collecting of the data to train the prediction model. The detailed procedure of the prediction technique is briefly discussed on the in the next section of the report.

4.3.1. Predictive technique to forecast workload based on SNR

The SNR measurement from the RAN side is used in our prediction techniques which is mainly on the random of the environment. Hence, the predictive technique must be able to forecast a random process whose output variable is of continuous value. The SNR can be measured at any time, which corresponds to a continuous-time random process. However, if the SNR measurement for a particular application with the 5G connectivity node is taken at equally spaced times, the predicted levels of the SNR can also be modelled as a discrete-time random process. We consider predictive techniques with the capability of forecasting several future time slots from a given moment. Therefore, we use the MDP and the sequential model as MDP for forecasting the random processes either in continuous-time domain or in discrete-time domain. In further step of the prediction techniques, we will utilize ML algorithm, e.g., reinforcement learning which is also the tuple of MDP that can improve the prediction accuracy of our problem.

4.4. AI image processing engine

AI image processing engine refers to the AI component dedicated for UC2 application. This goal of this component is the possibility of tracking objects among multiple frames. The “objects” to be tracked for this activity are identified as the road users and this objective can be archived by considering an AI state-of-the-art framework (i.e., Darknet) combined with some algorithms dedicated for road users tracking.
Before going into algorithm details, a review of the state-of-the-art methods for multiple object tracking is necessary. Object tracking is an application of deep learning where the program takes an initial set of object detections and develops a unique identification for each of the initial detections. Then, it tracks the detected objects as they move around frames in a video. In other words, object tracking is the task of automatically identifying objects in a images sequence and interpreting them as a set of trajectories with high accuracy. Often, there’s an indication around the object being tracked, for example, a surrounding square that follows the object, showing the user where the object is on the screen.
To associate a unique ID for each object in the scene, three methods are mostly considered. The first one is the mean shift method. It is similar to K-Means but replaces the simple centroid technique of calculating the cluster centers with a weighted average that gives importance to points that are closer to the mean. The goal of the algorithm is to find all the modes in the given data distribution. Also, this algorithm does not require an optimum "K" value like K-Means. Suppose we have detection for an object in the frame and we extract certain features from the detection (color, texture, histogram, etc). By applying the mean-shift algorithm, we have a general idea of where the mode of the distribution of features lies in the current state. Now when we have the next frame, where this distribution has changed due to the movement of the object in the frame, the mean-shift algorithm looks for the new largest mode and hence tracks the object.

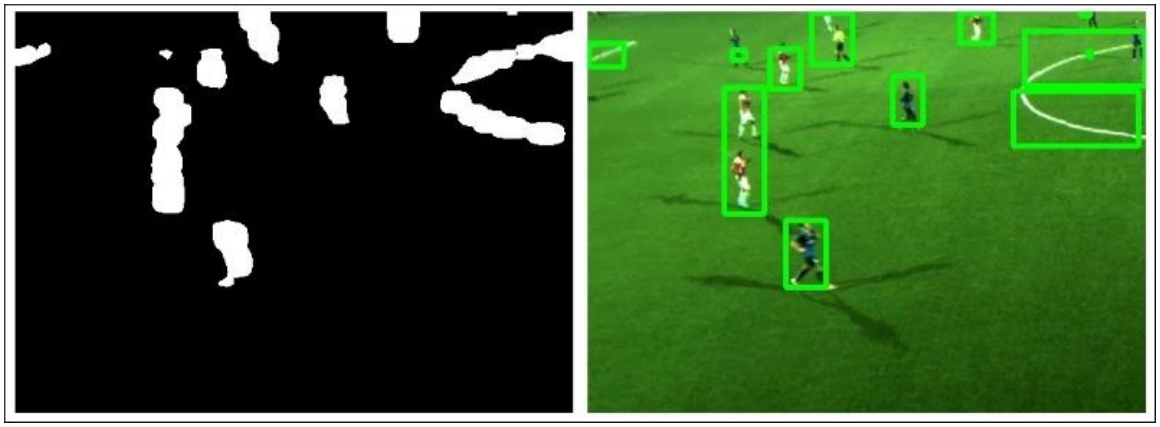

Figure 4.4: Example of mean-shift object tracking implemented in OpenCV

Another algorithm that is widely used is optical flow. This method differs from mean-shift, as we do not necessarily use features extracted from the detected object. Instead, the object is tracked using the spatio-temporal image brightness variations at a pixel level. Here we focus on obtaining a displacement vector for the object to be tracked across the frames. Tracking with optical flow rests on four important assumptions:

- Brightness consistency: Brightness around a small region is assumed to remain nearly constant, although the location of the region might change.
- Spatial coherence: Neighboring points in the scene typically belong to the same surface and hence typically have similar motions.
- Temporal persistence: Motion of a patch has a gradual change.
- Limited motion: Points do not move very far or randomly.

Once these criteria are satisfied, we use something called the Lucas-Kanade method to obtain an equation for the velocity of certain points to be tracked (usually these are easily detected features). Using the equation and some prediction techniques, a given object can be tracked throughout the video.

Figure 4.5: Example of Optical flow operation

The last approach which is commonly used is Kalman filtering. The core idea of a Kalman filter is to use the available detections and previous predictions to arrive at the best guess of the current state while keeping the possibility of errors in the process. For example, now we can train a good AI (for instance, Darknet) that detects a person. But it is not that accurate and occasionally misses detections, for instance 10% of frames. To effectively track and predict the next state of a person, let us assume a "Constant velocity model". Once we have defined the simple model according to laws of physics, we can make a nice guess on where the person will be in the next frame. However, we did not consider a noise component that is associated with the fact that we cannot always expect constant velocity and this noise is called "Process Noise". Moreover, the detector output is also not accurate in making predictions, thus we have "Measurement Noise" associated with it.

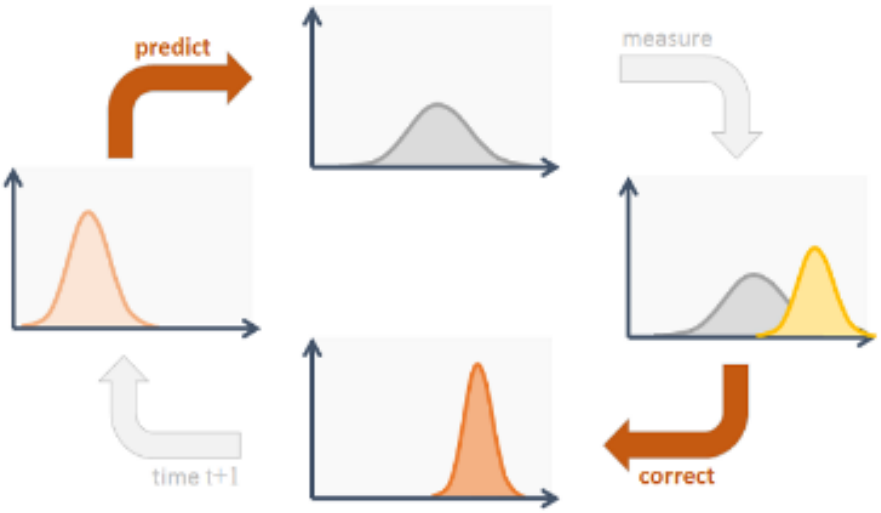

Figure 4.6: Kalman filtering workflow

As reported in Figure 4.6, the Kalman filter works recursively, where it takes current readings to predict the current state, then it uses the measurements to update the predictions. In other words, it creates a new distribution (the predictions) from the

previous state distribution and the measurement distribution. Kalman filter works best for linear systems with Gaussian processes involved. Road user tracking use case falls into the Gaussian realm, hence it is suited for the use of Kalman filters. For this reason, this last approach has been considered as AI image processing engine.

4.4.1. Learning Module

The learning module of the MOD application is used for learning state-of-the-art machine learning models. The learning module requires access to the influx DB, where found motifs by the Discovery module (WP4-T4.2) are stored. The detection models are learned from the discovered motifs, and a dictionary of models is created. The dictionary is then stored in the influx DB, and a copy is sent to the cloud. The Digital Twin utilizes the models' dictionary in the cloud to reconstruct the continuous data stream. The design of the data processing pipeline is depicted in Figure 4.7.

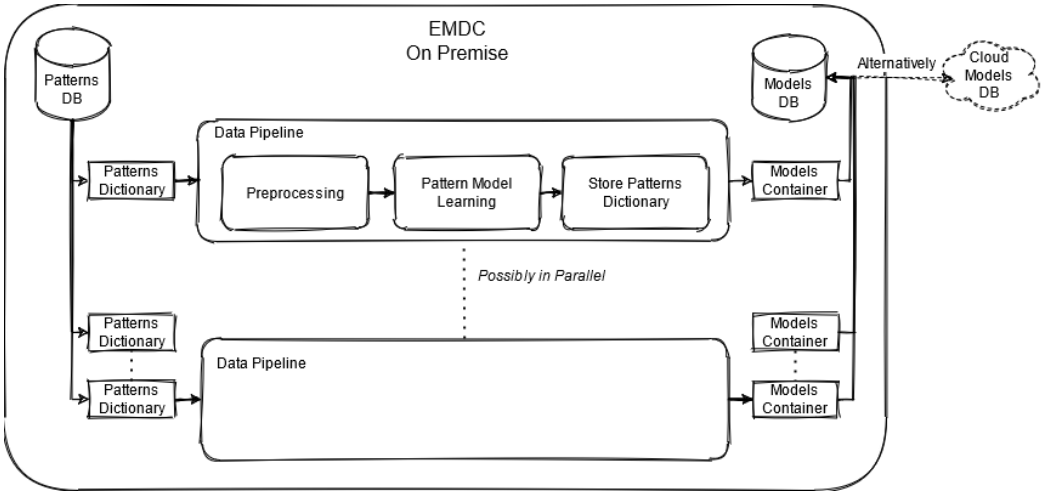

Figure 4.7: Learning module data processing pipeline.

5. Monitoring infrastructure

5.1. Network telemetry framework

5.1.1. Overview

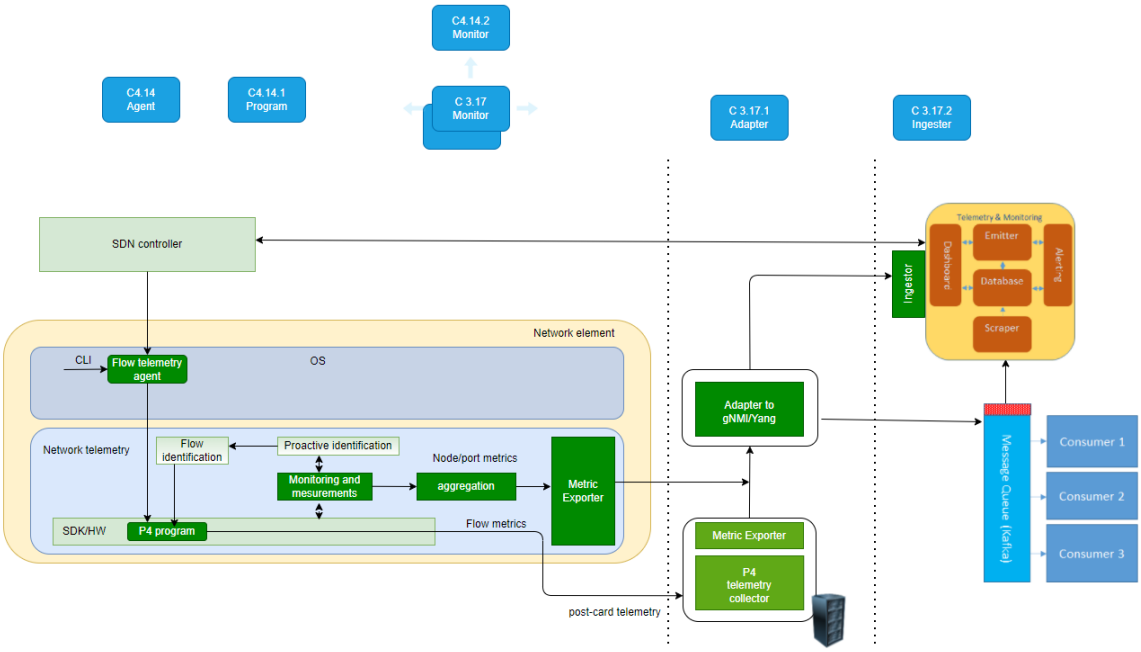

Figure 5.1 Network telemetry framework

Network telemetry framework purpose (see Figure 5.1) is to provide a real time information about current network status. This information is consumed later by multiple subsystems that include:

- Monitoring and alerting systems
- Network managers and controllers
- History collectors
- Etc.

So the framework should be able to receive and process multiple types of data with various characteristics and should have scalable approach in order to accommodate to different scale data centres.

The framework supports

- Flexible way to import different types of streaming telemetry from different platforms based on gRPC schemas supporting telemetry in different scales and granularities from per network node up to separate network flow
- Supporting both raw and aggregated data
- Flexible way to define attributes that express network state: permanent and transient
- Supports adding more attributes that can help telemetry consumers to identify data source, time, and location where data was originally produced
- Prepare data for export and consumption by one or multiple independent external collectors that are not familiar with specific data producer and operate in generic way

5.1.2. Components

The framework contains 3 components:

- Telemetry monitors
- Telemetry adapter
- Telemetry ingester

5.1.3. Telemetry monitor

Telemetry adapter is responsible to extract raw telemetry data from network element, process, translate it to generic format, optionally aggregate it and compress and stream to telemetry adapter system for further processing.

Telemetry monitor uses gRPC to stream the data to telemetry adapter, e.g. Figure 5.2

```
syntax = "proto3";
package braine_pb;
option go_package = ".;braine_pb";

message BraineAggregate {
        string hostname  = 1;
        string port      = 2;
        uint64 bandwidth = 3;
        uint64 timestamp = 4;
}

message BraineFlowSample{
        string hostname        = 1;
        string egress_port     = 2;
        string ingress_port    = 3;

        string sip             = 4 ;
        string dip             = 5 ;
        uint32 sport           = 6;
        uint32 dport           = 7;
        uint32 proto           = 8;
        uint32 buffer_occupancy = 9;
        uint32 latency         = 10;
        uint32 pkt_size        = 11;
        uint32 traffic_class   = 12;

        uint64 timestamp       = 13;
}
```

Figure 5.2 Monitor gRPC protocol

Every network node will run one or more telemetry monitors according network node capabilities and network requirements.

5.1.4. Telemetry adapter

Network telemetry adapter is responsible to get registrations from telemetry monitors, wait until one or multiple telemetry collectors connects to it and then starts to stream received data to collectors.

Telemetry adapter uses a YANG schema in order to inform collectors how data is going to look like, convert received telemetry data to the YANG format, and stream the data in that format using gNMI protocol.

When collector subscribes to receive a data it can decided to receive all the data or only subset of it based on the YANG schema that it received.

The example below Figure 5.3 presents the YANG schema that telemetry adapter uses in order to export the telemetry data. Curretnly it exposes two different substreams: aggregated and flow sample that are grouped per interface incoming interface, so subscribing collector can select only a specific interface to listen for.

```
module braine-telemetry {
        // Entrypoint /oc-if:interfaces/oc-if:interface
        //
        // xPath BW     --> interfaces/interface[name=*]/braine-telemetry/

        import openconfig-interfaces { prefix oc-if; }

        namespace "http://braine.com/yang/telemetry";
        prefix "braine-telemetry";

        revision "2021-12-27" {
                description
                        "Initial revision";
                reference "1.0.0.";
        }

        augment "/oc-if:interfaces/oc-if:interface" {
                uses interfaces-braine;
        }

        grouping interfaces-braine {
                description "Top-level grouping for BRAINE telemetry data.";
                container braine-telemetry{
                        container aggregated {
                                container state {
                                        leaf data {
                                                type string;
                                                description "Interface
Braine telemetry data  in JSON";
                                        }
                                }
                        }
                        container flow-sample {
                                container state {
                                        leaf packet-sample {
                                                type string;
                                                description "Packet sample
encoded in JSON";
                                        }
                                }
                        }
                }
        }
}
```

Figure 5.3 Telemetry hierarch in YANG model

The second table Figure 5.4 represents the data itself that is streamed. Collectors can extract this data according provided format and export it to other system.

```
module braine-types {
         namespace "http://braine.com/yang/telemetry-types";
         prefix "braine-telemetry-types";

         container aggregated {
                  description "Interface Braine telemetry data";
                  leaf port {
                           type string;
                           description "Port under measurements";
                  }
                  leaf bandwidth {
                           type uint64;
                           description "Port bandwidth";
                  }
                  leaf time{
                           type uint64;
                           description "Timestamp";
                  }
         }
         container packet-sample {
                  uses packet-info;
                  uses packet-telemetry;
         }
         grouping packet-info {
                  leaf sip {
                           type string;
                           description "Source IP";
                  }
                  leaf dip {
                           type string;
                           description "Destination IP";
                  }
                  leaf proto {
                           type uint32;
                           description "Protocol";
                  }
                  leaf sport {
                           type uint32;
                           description "Source port";
                  }
                  leaf dport {
                           type uint32;
                           description "Destination port";
                  }

         }
         grouping packet-telemetry {
                  leaf ingress-port{
                           type string;
                           description "Ingress port";
                  }
                  leaf egress-port{
                           type string;
                           description "Egress port";
                  }
                  leaf buffer-occupancy {
                           type uint32;
                           description "Buffer occupancy in units of 8KB";
                  }
                  leaf latency {
                           type uint32;
                           description "Latency in unit of 32 nanoseconds";
                  }
                  leaf pkt-size {
                           type uint32;
                           description "Packet size in Byte";

                  }
                  leaf traffic-class {
                           type uint32;
                           description "Traffic Class";
                  }
                  leaf time  {
```

Figure 5.4 Data representation YANG model

5.1.5. Telemetry ingester

Telemetry ingester acts as a subscriber to telemetry adapter – it subscribes to gNMI stream based on YANG schema that was received, starts to receive telemetry data and converts it to the format that is being used.

According the Braine architecture the telemetry data is received in InfluxDB.

The below example Figure 5.5 presents this:

```
[agent]
  interval = "10s"
  round_interval = true
  metric_batch_size = 1000
  metric_buffer_limit = 10000
  collection_jitter = "0s"
  flush_interval = "10s"
  flush_jitter = "0s"
  precision = ""
  hostname = ""
  omit_hostname = false
[[outputs.influxdb]]
   database="telemetry_mlnx"
   urls = ["http://localhost:8086"]
[[outputs.file]]
   files = ["/tmp/metrics.out"]
[[processors.parser]]
   parse_fields = ["data"]
   drop_original = false
   data_format = "json"
   tag_keys = ["Port"]
   json_string_fields=["Bandwidth", "Time"]
[[processors.parser]]
   parse_fields = ["packet_sample"]
   drop_original = false
   data_format = "json"
   tag_keys = ["Dip", "Dport", "EgressPort", "IngressPort", "Proto", "Sip",
"Sport", "TrafficClass"]
   json_string_fields=["BufferOccupancy", "Latency", "PktSize", "Time"]
[[inputs.gnmi]]
  addresses = ["localhost:9339"]
  encoding = "json"
   enable_tls = true
   insecure_skip_verify = true
   target = "braine"
[[inputs.gnmi.subscription]]
   name = "data"
   path = "/interfaces/interface[name=*]/braine-telemetry/state/data"
   subscription_mode = "target_defined"
 [[inputs.gnmi.subscription]]
   name = "packet-sample"
   path = "/interfaces/interface[name=*]/braine-telemetry/state/packet-
sample"
   subscription_mode = "target_defined"
```

Figure 5.5. Telegraf configuration

5.2. Telegraf agent for 5G Data collection and Collector and Forecasting Functional Block

5.2.1. Telegraf agent for 5G data collection

In this section, it will be reported the implementation details about the integration among FlexRAN and Kafka using Telegraf. As already presented, Mosaic5G FlexRAN is an SDN controller which implements RAN control interfaces on top of the Openairinterface code. It enables a Software Defined approach for external management capabilities by applications and third parties. This allows centralized and coordinated strategies to be applied among different base stations to improve spectrum efficiency and scale system capacity. The FlexRAN platform is composed of:

- FlexRAN Agents, which run on top of each BS;
- FlexRAN Real Time Controller (RTC), that interacts with and coordinates the agents.
- Both the RTC and agents have their own management modules, and exchange messages on top of a specific FlexRAN protocol.

In order to make FlexRAN possible, the original OAI-RAN code was extended to by-pass the original control plane and make it interact with a well-designed southbound API embedded within the agents. On the other side, FlexRAN RTC exposes a northbound API that allows applications to manage the RAN in an abstract manner.
The agent is also equipped with a set of control modules. Among them, there is the *Reports and event manager* module: it is used to notify the controller for available configurations/statistics reports that could be generated by a local event or an asynchronous request from the controller. Indeed, the controller implements certain endpoints within the northbound API which implicitly invokes the southbound one of its agents to retrieve the requested data.
One of the endpoints available at the controller is the */stats* one, which contains both static configurations (about BS, UE and LC) and statistics about many BS layers (PDCP, RLC and MAC). Since the idea is to use Kafka to optimize network configurations based on a stream of real time performance data, we focused on low-level statistics for UEs connected to the network. The mentioned endpoint returns data in Json format, which contains details for each UE attached to each BS.
Since those metrics require to be parsed and sent over a Kafka topic, a Telegraf agent has been configured for this purpose.
As already presented, Telegraf is a plugin-driven server agent used to collect metrics and to report events from different sources to different destinations. It offers a very low memory footprint, and a very good deployment flexibility. Plugins are mainly used to gather metrics from and send them to specific endpoints, but they can also aggregate or even pre-process them. For this specific case, the agent has been configured with an input plugin, called “execd”, which periodically invokes a Python script. The script retrieves the mac statistics through an http GET request to the FlexRAN */stats* endpoint. For each query, the script parses the returned JSON object and keeps only the information about each UE attached to the system. For each UE, it creates an ad hoc Json metric, which inherits any significant information from the original structure.

The output metrics follow the InfluxDB schema, using as tags the BS identifier and the UE identifier. Each of them is then sent, using a Kafka producer output plugin, over a dedicated Kafka topic. The system has been tested using a local cluster of three brokers, with strict ordering and redundant configurations.

The future plan are to integrate the Telegraf-based implementation with the BRAINE data collection framework to make data available for radio resource management purposes.

5.2.2. Forecasting module

A forecasting module has been developed by SSSA for forecasting parameters based on current and past measurement data. The forecasting module is based on both traditional statistical analysis techniques and AI/ML techniques.The current version of the module is a custom implementation to be applied to inband telemetry (INT) data collected from P4 switches and user equipment position.

The considered forecasting module, depicted in Figure 5.6, receives the time series of the computed delay value for the specific application and related user position from the switch responsible for traffic steering. Such data are used for both training the AI/ML-based forecasting algorithm and for forecast values inference. Note that different types of AI/ML techniques can be used, and training can be done in either the edge node or in the cloud if more computational resources are needed as reported in [Chin]. The forecast algorithm considered in this implementation is based on long short-term memory (LSTM). LSTM is a special form of a recurrent neural network (RNN) that can learn long-term dependencies based on the information gathered in previous steps of the learning process. LSTM consists of a set of recurrent blocks (i.e., memory blocks) where each block contains one or more memory cells and multiplicative units such as input, output, and forget gate.

LSTM is one of the most successful models for forecasting long-term time series. LSTM can be characterized by different hyper-parameters, specifically the number of hidden layers, number of neurons, and batch size. Details of LSTM parameters and their impact on prediction accuracy can be found in [Chau]. However, the process of finding optimal hyper-parameters that minimize the forecasting error could be time and resource consuming.

When LSTM is utilized for forecasting a time series, in general, the input vector/layer corresponds to the n previous data points, and the output vector/layer corresponds to k steps ahead with respect to the current time t of the considered time series. In this implementation, a stacked LSTM model is exploited with multi-step (i.e., k >1) forecasting.

In LSTM multi-step forecasting (LSTM-MSF), LSTM predicts k number of data points by considering n previous observed data points:

$$P(t+k, t+k-1, \ldots, t+1) = model(O(t), O(t-1), \ldots, O(t-n-1)), \quad (1)$$

where k > 1, P is the prediction of the single data point at time t, and O is the observed value at time t.

Note that offline training is considered in the evaluation, where weights are updated by using the backpropagation through time (BPTT) [Chau] gradient-based technique for training the data set. For more details on the considered implementation and performance evaluation the reader is referred to [Scan].

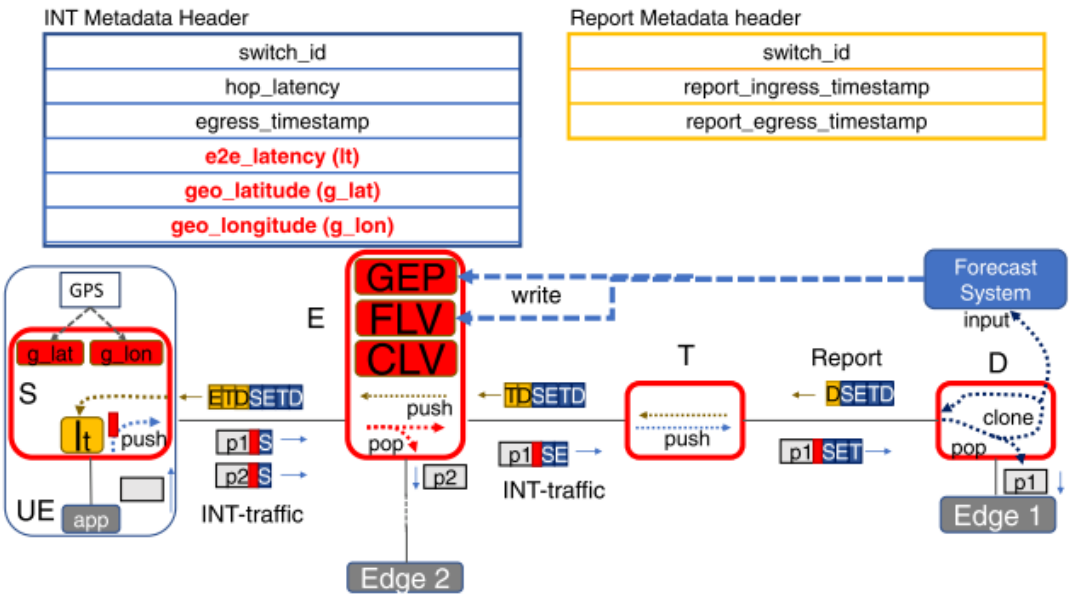

Figure 5.6: Forecasting module architecture

6. Components

All the components that are listed in MS8, outcome status sheet, and new ones.

Component ID	Component Name	Development	Owner
C3.1	BRAINE Service Mesh	90%	VMW
GitLab Repository: https://gitlab.com/braine/braine-mesh Containerized: Y Registered on BRAINE platform image registry: Y Deployed as a pod and functional on BRAINE platform: Y Integrated with other platform components: In progress This component is being integrated with UseCase 1 applications			
C3.13	Image Registry	95%	VMW
GitLab Repository: https://gitlab.com/braine/registry Containerized: N/A Registered on BRAINE platform image registry: N/A Deployed as a pod and functional on BRAINE platform: N/A Integrated with other platform components: Y			

Component ID	Component Name	Development	Owner
C3.5	BRAINE RL Scheduler	90%	LUH
GitLab Repository: https://gitlab.com/braine/wp3-work_placement-luh/ Containerized: Y Registered on BRAINE platform image registry: N Deployed as a pod and functional on BRAINE platform: Y Integrated with other platform components: Y. This component is integrated with the telemetry database via the Data Access Agent (of the cognitive framework). And also integrated with the ML-based inference engine for worker node selection via REST APIs			
C3.5.01	BRAINE RL Trainer	80%	LUH
GitLab Repository: https://gitlab.com/braine/wp3-work_placement-luh/-/tree/main/SchedulerTrainer Containerized: Y Registered on BRAINE platform image registry: Y Deployed as a pod and functional on BRAINE platform: Y Integrated with other platform components: Y. This component generates a trained ML model which is then persisted and utilized by the Inferencer (BRAINE RL Inference Engine for Scheduling)			
C3.5.02	Inferencer	70%	LUH

GitLab Repository: https://gitlab.com/braine/wp3-work_placement-luh/-/tree/main/SchedulerInference

Containerized: Y

Registered on BRAINE platform image registry: Y

Deployed as a pod and functional on BRAINE platform: Y

Integrated with other platform components: Y.

This component utilizes the trained ML model and serves the scheduling mechanism via a REST API for the prediction of a proper worker node for a given workload and system state.

C3.5.03	Data Access Agent	80%	LUH

GitLab Repository: https://gitlab.com/braine/wp3-work_placement-luh/-/tree/main/dataAcess

Containerized: Y

Registered on BRAINE platform image registry: Y

Deployed as a pod and functional on BRAINE platform: Y

Integrated with other platform components: Y.

This component acts as a bridge between the scheduling plugin and the telemetry database. Its responsibility is to serve the plugin, via a REST API, with information about the resource usages/availability of each of the worker nodes of the cluster.

Component ID	Component Name	Development	Owner
C3.6	Telemetry Infrastructure	85%	LUH

GitLab Repository: https://gitlab.com/braine/wp3-telemtry-luh/-/tree/main/T34/telemetry

Containerized: Y

Registered on BRAINE platform image registry: Y

Deployed as a pod and functional on BRAINE platform: Y

Integrated with other platform components: Y

Status Report:

The telemetry infrastructure consists of multiple components are integrated, dockerized, podified, and, and deployed. The components are:

- C3.6: Telemetry database using InfluxDB
- C3.6.01: Telemetry metric exporter using node_exporter, cAdvisor, and use-case applications
- C3.6.02: Telemetry scraper using Prometheus
- C3.6.03 Telemetry Alerting using Alert Manager (under research and test)
- There is also a monitoring dashboard but the component is realized as a part of WP4 (T4.4)

Component ID	Component Name	Developme	Owner

		nt	
C3.8	MOD – Learning module	90%	FS
GitLab Repository: https://gitlab.com/braine/wp3-mod_learning_module-fs Containerized: Y Registered on BRAINE platform image registry: Y Deployed as a pod and functional on BRAINE platform: Y Integrated with other platform components: Y – Discovery Module and Detection Module of MOD Application Status Report: Learning module was tested on CNIT Braine Testbed. Currently this component is being integrated within the UC3 and other modules of Motif Discovery Tool (WP4).			

Component ID	**Component Name**	**Development**	**Owner**
C3.9	Image Orchestrator	100%	ECC
GitLab Repository: https://github.com/eccenca/braine/tree/main/container-orchestrator Containerized: N Registered on BRAINE platform image registry: N Deployed as a pod and functional on BRAINE platform: N Integrated with other platform components: Y – The Image Orchestrator performs image deployment and synchronize metadata between the Global Image Registry and the BRAINE Image & Service Catalog. Status Report: The Image Orchestrator is fully functional and integrated and with the Global Image Registry.			

Component ID	**Component Name**	**Development**	**Owner**
C3.10	Metrics Relay	100%	ECC
GitLab Repository: https://github.com/eccenca/braine/tree/main/relay Containerized: N Registered on BRAINE platform image registry: N Deployed as a pod and functional on BRAINE platform: N Integrated with other platform components: Y – The relay collects Nodes' metadata from the SLA Broker and populate BRAINE catalog with available Nodes and resources. Status Report: The Metrics Relay is fully developed and integrated with the SLA Broker.			

Component ID	**Component Name**	**Development**	**Owner**

C3.11	BRAINE Ontology	100%	ECC

GitLab Repository: https://github.com/eccenca/braine-vocab

Containerized: N

Registered on BRAINE platform image registry: N

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: Y– The BRAINE ontology is being used to instantiate the BRAINE Resource & Service Catalog.

Status Report:

The BRAINE Ontology has been created and is being periodically improved with revisions addressing issues.

Component ID	**Component Name**	**Development**	Owner
C3.12	Knowledge Bootstrapper	100%	ECC

GitLab Repository: https://github.com/eccenca/braine/tree/main/setup

Containerized: N

Registered on BRAINE platform image registry: N

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: Y – Integrated with the BRAINE Resource & Service Catalog.

Status Report:

The Knowledge Bootstrapper has been created, tested, and is being used to bootstrap information at the Resource & Service Catalog with BRAINE data model.

Component ID	Component Name	Use Cases	Owner
C3.13	SDN Controller	UC2	CNIT

GitLab Repository: https://gitlab.com/braine/WP3-SDN-CONTROLLER

Containerized: N

Registered on BRAINE platform image registry: N

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: Y – Info

The SDN controller will configure the network layer aiming at both enabling the traffic forwarding (with the required QoS) and the traffic monitoring toward the telemetry system. In its first version the BRAINE SDN controller was released with a northbound application (i.e., the BRAINE app) opening a REST APIs toward the K8s orchestrator and other BRAINE components. In this period a companion application has been developed, tested, and demonstrated in an international conference to enable the discovery of PODs deployed by K8s and the forwarding and monitoring of traffic exchanged among PODs.

With this additional application the SDN controller enables the matching of the traffic at the PODs level (assuming K8s working with the Flannel tool in VXLAN mode) thus enables the specific monitoring of each traffic flow with the required granularity. This enables the detection of QoS degradation (e.g., latency degradation) by the telemetry system that can provide feedback to the SDN controller itself to take actions on the

network aiming at recovering the QoS requirements satisfaction.

The SDN controller is integrated with the telemetry system in both directions. Indeed, the SDN controller is able to configure the network devices to forward postcard telemetry data toward a telemetry collector point that, after some aggregation, pushes the data into the InfluxDB. In turn, the telemetry system exploiting the Graphana tool is able to generate alarms that triggers a call to the REST APIs of the SDN controller itself.

The integration with the K8s orchestrator is in phase of development. The SDN orchestrator and the P4 application have been extended to allow matching and telemetry of the traffic at the POD level. Moreover, an interaction with the K8s REST has been designed to import detailed information regarding deployed PODs.

Status Report:

The SDN controller is based on the ONOS open-source project. The additional ONOS components developed for BRAINE (i.e., the BRAINE app and the P4 app have been updated to the BRAINE GitLab). Both applications are in phase of testing on the CNIT testbed, therefore they may be improved to introduce additional functionalities and to fix possible issues.

Component ID	Component Name	Development	Owner
C3.16	Multi-access Edge Computing platform	100%	SMA

GitLab Repository: https://github.com/Sma-RTy/native-on-prem.git

Containerized: N

Registered on BRAINE platform image registry: N

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: N – Alternative platform for performance comparison

Status Report:

MEC platform is online and available to host third parties' applications for UC2

Component ID	Component Name	Development	Owner
C3.17	AI image processing engine	70%	SMA

GitLab Repository: https://github.com/Sma-RTy/deepsort

Containerized: Y

Registered on BRAINE platform image registry: Y

Deployed as a pod and functional on BRAINE platform: Y

Integrated with other platform components: Y – Application to be deployed in BRAINE Kubernetes cluster

Status Report:

The AI for tracking road users has been developed and a Docker container has been created. The application has to be ported to BRAINE platform and deployed in a Kubernetes cluster with hardware acceleration support.

Component ID	Component Name	Development	Owner
C3.18	Edge-to-edge multiagent communication	75%	CTU

GitLab Repository:

Containerized: N

Registered on BRAINE platform image registry: N

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: N – but there is a plan to utilize RabbitMQ as the MQTT broker for the MOD component

Status Report:

After testing the first version of the prototype of multiagent communication, architectural changes were introduced to simplify and speed up the communication. The whole architecture uses RabbitMQ as an AMQP broker that passes messages to particular agents. At the same time, FIPA inspired schema of the message, and asynchronous message processing was introduced to allow multiagent negotiation.

The component is currently in the stage of incorporation into the agents. As a further step, the communication needs to be tested with the whole multiagent platform and, based on the results, modified and optimized if necessary.

Component ID	**Component Name**	**Development**	**Owner**
C3.17	Telemetry monitors & exporter	35%	MLNX

GitLab Repository:

Containerized: Y

Registered on BRAINE platform image registry: N

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: Y

Status Report:

After several prototypes we finished initial implementation of the telemetry monitor application that includes a generation of telemetry data and streaming it to any addressable telemetry adapter.

On the next phase we plan to support information extraction from network device drivers that will include bandwidth and latency

Component ID	**Component Name**	**Development**	**Owner**
C3.17.1	Telemetry adapter	70%	MLNX

GitLab Repository:

Containerized: Y

Registered on BRAINE platform image registry: N

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: Y

Status Report:

After several prototypes initial implementation specific for currently exported telemetry data and YANG model is finished. It includes basic bandwidth and flow telemetry measurements.

In the next phase the support for additional telemetry like latency histograms will be introduced – it will include south interface based on gRPC for telemetry streamers and northbound interface for YANG subscribers.

It is also planned to podify the container to simplify its integration and deployment in BRAINE cluster.

Component ID	**Component Name**	**Development**	**Owner**
C3.17.2	Telemetry ingester	85%	MLNX

GitLab Repository:

Containerized: Y

Registered on BRAINE platform image registry: N

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: Y

Status Report:

After several prototypes it was decided to implement telemetry ingester with Telegraf framework. And Initial implementation specific for currently exported telemetry data and initial InfluxDB schema is finished.

In the next phase we plan to expand the implementation for final telemetry parameters and based on telemetry consumers' feedback update the exporting schema to enable better performance and visualization.

Component ID	Component Name	Development	Owner
C3.3	Telegraf agent for 5G data collection	80%	SSSA

GitLab Repository: https://gitlab.com/braine/wp3-5g-sssa/-/tree/main/T31

Containerized: Y

Registered on BRAINE platform image registry: Y

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: Y - Forecasting functional block

Status Report:

The collection module is currently functional and perfectly integrated with the rest of the platform. The next step is to check the correctness of the developed Pod manifest.

Component ID	Component Name	Development	Owner
C3.5	Forecasting functional block	60%	SSSA

GitLab Repository: https://gitlab.com/braine/wp3-ffb-5g-sssa

Containerized: N

Registered on BRAINE platform image registry: N

Deployed as a pod and functional on BRAINE platform: N

Integrated with other platform components: Y – Telegraf agent for 5G data collection

Status Report:

A preliminary implementation of this module has been developed. It takes in input data from a Kafka topic and, using an LSTM model, produces forecasted data to another Kafka topic.

Component ID	Component Name	Use Cases	Owner
C4.10	Exporter for the metrics for the UC1 application 'AI-driven Digital Twin solution for new digital ecosystems enabling Smart Healthcare in Medical and Caregiving Centres'	UC1	IMC

The 'AI-driven Digital Twin solution for new digital ecosystems enabling Smart Healthcare in Medical and Caregiving Centres', being part of a WP5 as Use Case1 and is also part of the WP3, WP4 where specific work is being carried out e.g. adaptation the Edge-based system for human-centric applications.

IMC has prepared the 'BRAINE Living eHealth model' (part of WP3 task plan) focused on the healthcare-specific environment with the consideration that application shall be operated and run on EMDC with the clear understanding what is actually happening to the application itself.

While the Telemetry Infrastructure (component C3.6) is focused on the EMDC itself, additional component was designed and develop for the 'AI-driven Digital Twin solution for new digital ecosystems enabling Smart Healthcare in Medical and Caregiving Centres'.

Although the component is registered in WP4, it is an integral part of this work package (WP3) and custom metrics to get meaningful data about application performance were designed. The exporter for the metrics for the UC1 application connects to the C3.6 and provides an endpoint "/metrics" and sends GET metrics on request from the Prometheus server and deployed as a pod and is functional on BRAINE platform (Y)/

7. Conclusion

This deliverable has presented the main activities in year-2 of the BRAINE project related to the design, prototype and implementation of the BRAINE WP3 components. The deliverable dedicated a specific section for each task to show its main contributions. The illustrated achievements include components functionalities and development status. Moreover, the design of a novel Cognitive Framework is described in this document. Finally, a list of all WP3 software components' details and links to their implementations in the BRAINE Gitlab account is also provided.

8. References

[Chau]: Y. Chauvin and D. E. Rumelhart, eds., Backpropagation: Theory, Architectures, and Applications (L. Erlbaum Associates, 1995).

[Chin]: V. R. Chintapalli, K. Kondepu, A. Sgambelluri, A. Franklin, B. R. Tamma, P. Castoldi, and L. Valcarenghi, “Orchestrating edge-and cloud-based predictive analytics services,” in European Conference on Networks and Communications (EuCNC) (2020), pp. 214–218.

[Chon]: Chongjin Xie, et al, “Open and disaggregated optical transport networks for data center interconnects [Invited]” JOCN 2020.

[Cugi]: F. Cugini, D. Scano, A. Giorgetti, A. Sgambelluri, P. Castoldi, F. Paolucci, “P4 programmability at the network edge: the BRAINE approach”, ICCCN 2022, Athens, Greece.

[Ferr]: A. Ferrari et al.,”GNPy: an open source application for physical layer aware open optical networks”, JOCN 2020.

[Gior]: A. Giorgetti, et al., “Control of open and disaggregated transport networks using the Open Network Operating System (ONOS)”, JOCN 2020.

[Hern]: J. Hernandez, et al, “Comprehensive model for technoeconomic studies of next-generation central offices for metro networks”, JOCN 2020.

[Lope]: V. Lopez, et al., “Enabling fully programmable transponder white boxes [Invited]”, JOCN 2020.

[Mans]: C. Manso et al., "TAPI-enabled SDN control for partially disaggregated multi-domain (OLS) and multi-layer (WDM over SDM) optical networks," JOCN 2020.

[Paol]: F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, and P. Castoldi, “P4 Edge Node enabling Stateful Traffic Engineering and Cyber Security,” IEEE/OSA Journal of Optical Communications and Networking, vol. 11, no. 1, pp. A84–A95, Jan. 2019.

[Ricc]: E. Riccardi, et al., “An Operator view on the Introduction of White Boxes into Optical Networks”, JLT 2018.

[Scan]: Davide Scano, Francesco Paolucci, Koteswararao Kondepu, Andrea Sgambelluri, Luca Valcarenghi, and Filippo Cugini, "Extending P4 in-band telemetry to user equipment for latency- and localization-aware autonomous networking with AI forecasting," J. Opt. Commun. Netw. **13**, D103-D114 (2021)

[Sgam-1]: A Sgambelluri, A Giorgetti, D Scano, F Cugini, F Paolucci, “OpenConfig and OpenROADM Automation of Operational Modes in Disaggregated Optical Networks”, IEEE Access 2020.

[Sgam-2]: A. Sgambelluri, et al, “Coordinating Pluggable Transceiver Control in SONiC-based Disaggregated Packet-Optical Networks”, OFC 2021.