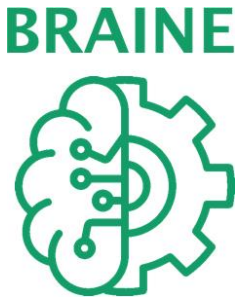

BRAINE - Big data Processing and Artificial Intelligence at the Network Edge

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: D3.4

Fourth report on the status of WP3

Due date of deliverable:	30 November 2022
Actual submission date:	28 February 2023
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	Final project report on the status of WP3 – Part 2
Deliverable number	D3.4
Deliverable version	Version 1.0
Previous version(s)	-
Contractual date of delivery	30 November 2022
Actual date of delivery	28 February 2023
Deliverable filename	Final project report on the status of WP3 – Part 2
Nature of deliverable	Report
Dissemination level	PU
Number of pages	48
Work package	WP3
Task(s)	T3.1, T3.2, T3.3, T3.4
Partner responsible	DELL
Author(s)	Mustafa Al-Bado (Dell), Alessio Giorgetti (CNIT)), Javad Chamanara (LUH), John Rothman (LUH), Edgard Marx (ECC), Luca Valcarenghi (SSSA), Alessandro Pacini (SSSA), Andrea Sgambelluri (SSSA), Emilio Paolini (SSSA).
Editor	Mustafa Al-Bado (Dell)

Abstract	
Keywords	

Copyright

Deliverable history

Version	Date	Reason	Revised by
00	08.11.2023	Table of Contents - version 00	Mustafa Al-Bado
01	17.02.2023	Final review	Mustafa Al-Bado

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

This report provides an update on the developments made in year-3 of the BRAINE project related to the design, prototype, and implementation of the BRAINE WP3 components. This technical report presents the final outcome of WP3 (Part 2), which includes several sections and results.

Specifically, the current deliverable highlights the following:

- An update on the developments made in year-3 of the BRAINE project related to the design, prototype, and implementation of the BRAINE WP3 components.
- The design of a novel Cognitive Framework and provides a list of all WP3 software components' details and links to their implementations in the BRAINE.
- A recap of the architecture of the Forecasting Functional Block (FFB) and describes its flexibility in terms of models and metrics.
- Highlights the challenges in deploying K8s in edge computing environments due to bandwidth limitation or bounded latency. The report presents a specifically designed and comprehensive framework to address these challenges that relies on SDN network controller, Service Level Agreement (SLA) broker, and Telemetry Collector.
- The extension of the vocabulary to support workflow placement and description.
- The design and implementation of the SLA Broker and its role in resolving system violations.

Table 1.1 lists the components reported in D3.4 including partners, components' names and Figure 1.1 shows an architecture diagram of where these components integrate as part of the overall BRAINE.

Partner	Components	Deliverable
LUH	RL Scheduler - Training Agent (C3.6.1)	D3.4
	RL Scheduler - Inference Engine (C3.6.2)	
	RL Scheduler – K8s Scoring Plugin (C3.6.4)	
SSSA	Forecasting functional block (C3.24)	D3.4
CNIT	SDN network controller (C3.13)	D3.4
ECC	Image Orchestrator (C3.11)	D3.4
	BRAINE Schema for describing Services & Computational Resource (C3.23)	
DELL	SLA Broker (C3.15)	D3.4

Table 1.1: Reported components

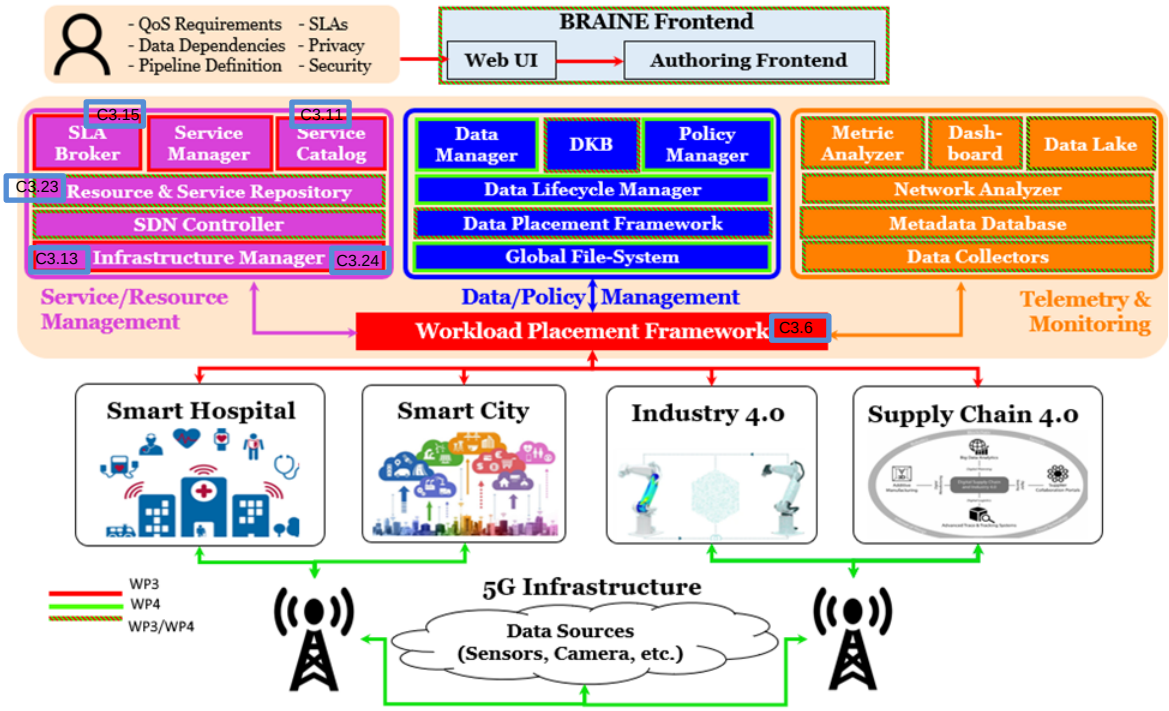

Figure 1.1: An architecture diagram of where components integrate as part of the overall BRAINE

2. AI-based workload placement in an edge environment

The overall architecture and the detailed description of the AI-based workload placement were presented in D3.2 last year. Here in this report, a short recap of the architecture is provided. Then, this document describes the data and the performance aspects of the work.

2.1. AI/ML-based scheduler

The BRAINE scheduler (available at: https://gitlab.com/braine/wp3-work_placement-luh/) customizes the default behavior of the Kubernetes scheduler by using deep reinforcement learning (DRL) in the node scoring step to optimize the node selection strategy for energy or waiting time reduction. 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 selected 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. A high-level illustration of the different components involved in the proposed RL-based scoring plugin is presented in Figure 2.1.

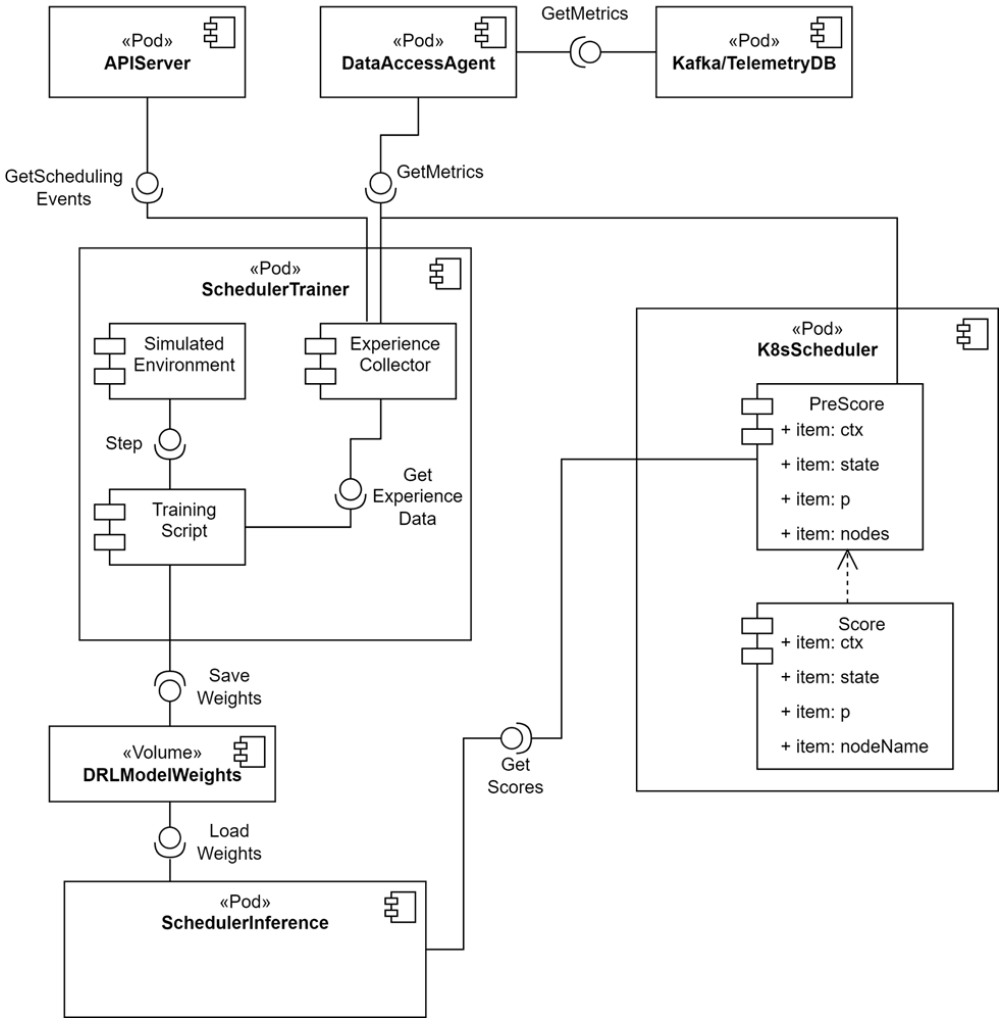

Figure 2.1: Component diagram of BRAINE RL scheduler

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, workload types, and optimization objectives.
2. **Scheduler Inference**: is a containerized RESTful API 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.
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.
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.

2.2. State and Model Description

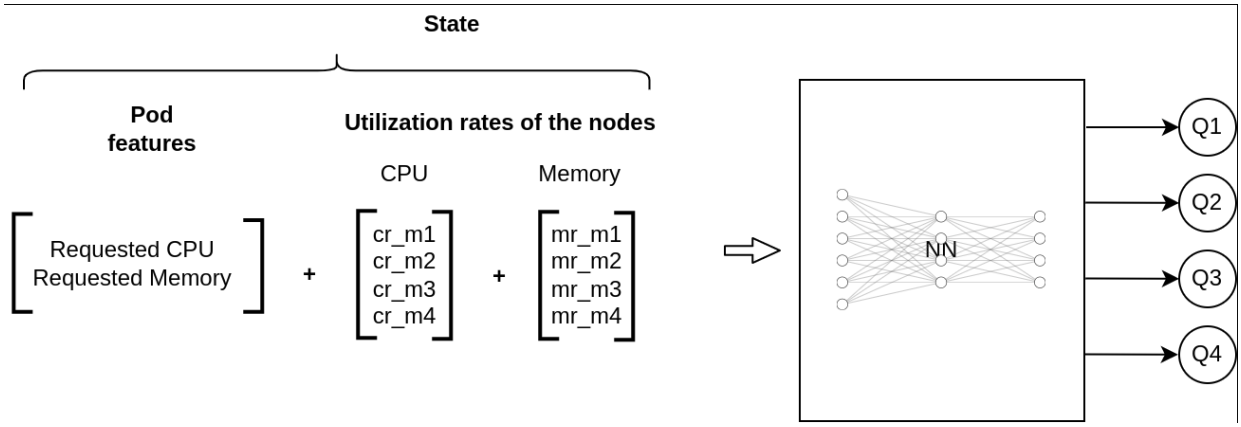

Figure 2.2: Representative state vs. Neural Network output

The representative state is shown on the left-hand side, while the Neural Network output is shown on the right-hand side of in Figure 2.2. The state is comprised of CPU request and Memory request as well as the Utilization rates for the resources on each node. The output score Q1 is Score of Machine 1, Q2 is the score for Machine 2, etc. The machine with the highest score is the chosen pod.

2.3. State space Rendering with multiple policies

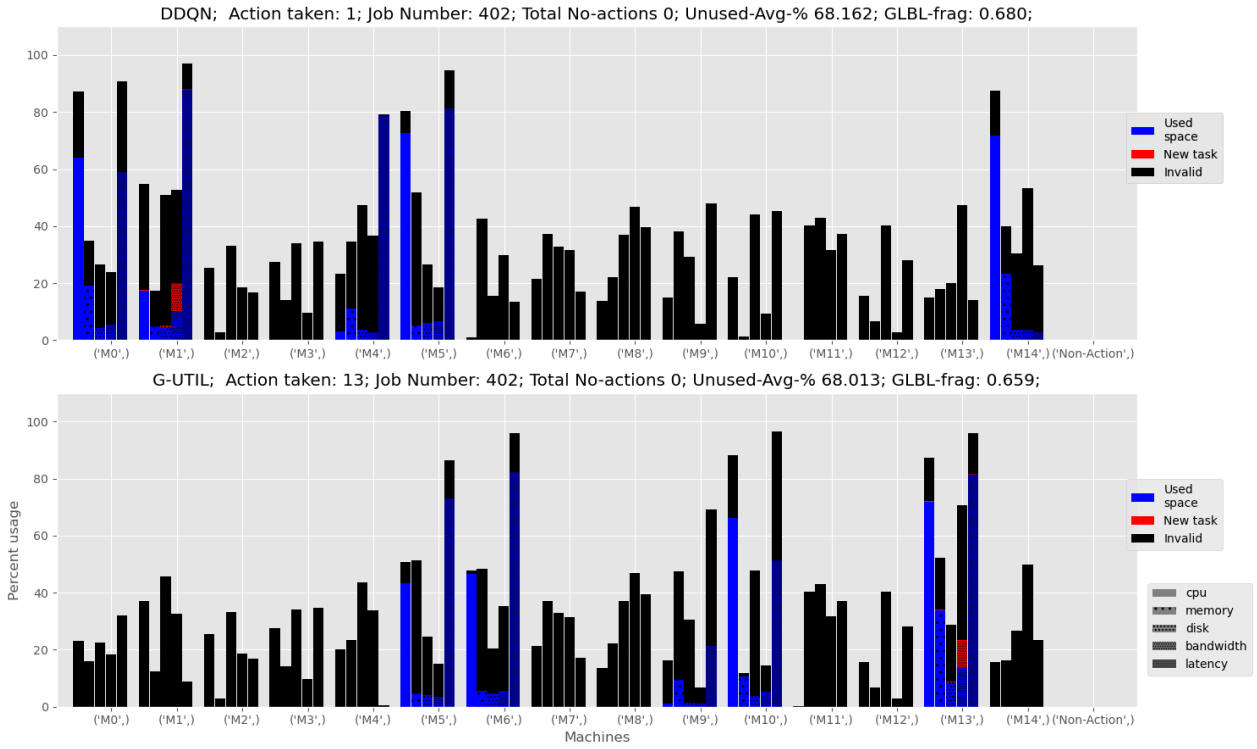

Figure 2.3: The trained model policy (top) vs. graph represents a greedy Utilization policy (bottom)

In this example we are using 5 resource dimensions (CPU, memory, disk, bandwidth, and latency). The y-axis represents the usage of each resource dimension. Figure 2.3 represents the trained model policy. while the bottom graph represents a greedy Utilization policy.

The black bars represent a part of a machine which is inactive. The idea here is that we have some maximum resource value available, and all machines will take up some percentage [100%,0%) of each resource dimension. During training and testing each machine has each resource capacity randomly generated.

The blue bars represent currently used up machine usage. Note that a fully utilized machine will have the black bar plus the blue bar at 100%. Also, the red bar represents a newly placed pod.

2.4. Dataset Description

2.4.1. Definitions

Table 2.1 lists the definitions used in the explanation of the training data.

Term	Definition
Resource Utilization	Here we are only considering normalized resource utilizations. 1 is maximum, 0 is minimum.
R∈R	Resource “r” is an element out of the set of resources “R” One example of “R” could be [CPU, DISK, MEMORY]

Spiked Resource: SR{r}	A single resource chosen out of the set “R” which by definition will have a resource request and utilization which is significantly greater than the other resource dimensions
SRP{r}	Spiked Resource probability for resource “r”. This would indicate the probability of selecting a specific resource.
Non spiked resources: NSP	I’ll use this to denote all resources which are not spiked. So if Spiked Resource is denoted as SR{r}, All non-spiked resources is denoted as NSP{R\r}
Spiked Pod : SP{r}	SP{r} = Pod with resource dimension utilization “r” spiked. A spiked resource dimension will have a usage much greater than the others, aka. Spiked resource. So a pods spiked resource might have 5-20 times more utilization across its dimension when compared to all other dimensions.
Spiked Pod set : SPS{r}	SPS{r} = A group of Spiked Pods where all pods have the same spiked resource dimension “r”. This can vary in length. E.g. A SPS length of 20 could give 20 pods with the CPU resource dimension at 5-20% utilization, while Disk and Memory will both have 0-1% utilization. Currently we are only training/testing with fixed SPS lengths.
Pod Length Mu	Value to control the Pod Length normal distribution mean value. The Pod Length determines how long a pod lasts.
Pod Length Sigma	Value to control the Pod Length normal distribution spread value

Table 2.1: Definitions used in the explanation of the training data

Utilization magnitude: All resource dimensions are normalized to the interval of [0 .. 1]. Each SP{r} (Spiked Pod for a specific resource) has a single SR (spiked resource), and many NSR’s (non-spiked resource). The NSR’s have an independently and identically distributed random (IID) selection in the range of [0, 0.02], or [0, 0.01] depending on the test. In other words, up to 2% or 1% utilization, respectively. The spiked resource will have an identically distributed (ID) random selection in the range [0.05, 0.2].

Pod Durations: The Pod Durations are generated with a normal distribution with $\boldsymbol{\mu}$ (Pod Length Mu) and $\boldsymbol{\sigma}$ (Pod Length Sigma). The $\boldsymbol{\mu}$ value controls the normal distribution mean value, while the $\boldsymbol{\sigma}$ value controls the normal distribution spread value. Pod durations below 1 second are rounded up to 1 second.

Training Random selection with removal: This ensures the least amount of overlap of spiky resources when iterating through the SPS. Assuming S is a copy of the set R, Procedurally for each SPS{r} we randomly select an element from the set of resources in S with removal. This is continued over all resources until S is empty. In which we reset S by performing S=R and continue back to random element selection with removal. This can also be thought of as having a list of all resources, and picking out (and not putting it back) one at a time to use as the Spiky Resource in the SPS. When your list is empty, just refill it and start over.

Testing Random selection without removal: Doing Random selection with removal is not a realistic situation so for testing we must use a different strategy. We have some Test results experimenting with Random element selection from R without removal. Therefore for each SPS{r} we randomly select an element from the set of resources in S without removing resources from the set.

Resource probability selection (for without removal): The probability of a specific resource being selected for a SPS can be controlled so give weights to specific resources. This enables us to test in environments that might have a single specific resource as a SR, e.g. mainly CPU intensive environments.

We will denote the SPS's SR (Spiked Pod Set's Spiked Resource) probability of being selected as RESOURCE(PROBABILITY), e.g. CPU(0.5) would represent that for each new SPS, there is a 50% chance that the CPU will be selected as the SR. All probabilities must sum to one, so the "leftover probabilities" are evenly given to all the other resource dimensions. We can also represent multiple resources, e.g. CPU(0.2), DISK(0.3) which translates to a 20% probability of selecting CPU as the SR, and 30% probability of selecting DISK as the SR.

SPS example for understanding: Example 2-1 illustrates examples of training with Random resource selection for each Spiked Pod Sets (SPS), but with removal. Given 3 resource dimensions (CPU, Memory and Disk), and with SPS length of 4, that would mean that after 12 pods (4*3) we will have fully cycled through all resource dimensions as a Spiked Pod Resource. Below we will illustrate one full training iteration through all resources, but with grouping the pods by SPS

First Randomize resources -> [CPU, MEM, DISK]

SPS set name	Description
A	Train SPS{CPU} =Train with the 4 pods with SP{CPU}
B	Train SPS{MEM} =Train with the 4 pods with SP{MEM}
C	Train SPS{DISK} =Train with the 4 pods with SP{DISK}

Example 2-1: Randomize resources -> [CPU, MEM, DISK]

Now that we iterated though all resource we will randomize the resources again -> [MEM, CPU, DISK] (see Example 2-2)

SPS set name	Description
D	Train SPS{MEM} =Train with the 4 pods with SP{MEM}
E	Train SPS{CPU} =Train with the 4 pods with SP{CPU}
F	Train SPS{DISK} =Train with the 4 pods with SP{DISK}

Example 2-2: Randomize resources -> [MEM, CPU, DISK]

Table 2.2 visualizes SPS A, B, and C without grouping of each SPS. Spiked resources are highlighted.

Below we illustrate the first table but without the SPS groupings so we can see each individual pod along with sudo-generated utilizations for each resource.

SPS set name	Spiked Resource	CPU utilization %	MEM utilization %	DISK utilization %
A	CPU	8	1	1.1
A	CPU	5	1.5	0.1
A	CPU	15	2	0.8
A	CPU	19	0.5	0.1
B	DISK	0.9	0.01	6
B	DISK	1.7	0.8	17
B	DISK	0.7	1.2	11
B	DISK	1.01	0.9	5
C	MEM	1.1	7	0.4
C	MEM	1.9	17	0.7
C	MEM	0.9	10	1.7
C	MEM	0.4	20	0.9

Table 2.2: illustration of SPS A, B, and C without grouping of each SPS. Red indicates the spiked resource dimension.

We repeat this process until the dataset is complete or training has terminated. We then randomize all training/testing pod requests, as well as the pod duration lengths, while maintaining the SPS structure.

2.5. Performance of the Optimization Objectives

2.5.1. Definitions

Table 2.3 lists the definitions used in the explanation of the reward calculation.

Rewards		
Rewards for Increasing Pod Throughput	Fragmentation Average	Reward that calculates the fragmentation average score across all machines. Modification from the paper “Scheduling of Time-Varying Workloads Using Reinforcement Learning”
	Simple constant	Every job scheduled just returns a plain 1. The learning happens from the agent trying to squeeze more jobs in the machine before the episode ends
Rewards for Decreasing Energy Consumption	Utilization Spread	We take the utilization reward **utilization = np.sum(np.power(usages, 3)) / len(usages)** And Divide by the number of machines being used

	utilization / number_of_machines_being_used This will be similar to utilization reward, but will have a stronger punishment for using more machines.
Machine Job Fraction Reward	**np.log(Total Jobs Running now+ 1) / (Number of Machines used)**
Min Machine Reward	**(Number Machines Unused)/ (Total Number of Machines)**
Negative Machine Reward	**((Total Number of Machines) - (Number Used Machines))/(Total Number of Machines)** This reduces the reward as more machines are used, but it is never negative.

Table 2.3: Definitions used in the explanation of the reward calculation.

Total Pods Before Full Cluster (∝Reduced Pod wait time): For now the metric we are monitoring is "Total Pods Before Full Cluster," (TP). This metric is going to pre proportional to the following metrics: Throughput, Fragmentation Score, and Reduced Pod Wait Time. . Depending on what metric we want to show, the current TP metric can be transformed into any other metric if given the correct coefficients. For now I'll show the TP metric, but later we can easily change it to the "Reduced pod wait time,"

Workload: Workload type is similar to batch processing. In this case we are simply testing the efficiency of how the model stacks a given series of pods.. We are testing with long running tasks. While we are ending an episode as soon as the machine is full, if we were to account for pods waiting to be submitted then we could determine reduced pod wait time.

Episode termination: An episode is terminated if the machine fills up, and cannot place the next pod. This is helpful for the model to learn, because when combined with Prioritized Experience Replay (PER) it will tend to learn from events that had higher rewards, i.e. episodes where the policy stacked the pods more effectively. The task durations are set to infinite because we are highlighting the models ability to stack the pods in a better way when compared to the K8S-MA (Kubernetes most action, aka Greedy-Utilization).

2.5.2. Optimization Objective: Waiting Time

Run Parameters:

- Spiked Resource (SR) Random Selection from Uniform Distribution (0.05,0.2]
- Non Spiked Resource (NSR) Random Selection from Uniform Distribution (0.00,0.02]
- Spiked Pod Set (SPS) length 40
- 15 machines
- 5 resource dimensions
- Learning Rate 1e-4
- Long Pod durations (no pods are removed during training or testing, only continuous stacking of pods)
- 200 Test instances

Percent improvement from Greedy Utilization							
Row ID	**Random Selection Process**	**SPS Resource Selection Distribution**	**Policy Simple constant**	**Policy Frag Average Reward**	**Policy Utilization Fraction**	**Policy Min Machine**	**Policy Machine Job Fraction**
1	With Removal	Not a uniform distribution	26.87	3.22	1.54	2.26	1.80
2	With replacement	Uniform	19.5	2.73	1.47	2.05	1.71
3		CPU(0)	8.10	1.32	0.57	0.57	0.76
4		CPU(0.5)	8.11	1.35	0.44	0.95	0.64
5		CPU(0.8)	0.10	0.20	0.14	-0.01	0.32
6		CPU(1)	-1.1	0.08	0.19	-0.03	0.22

Table 2.4: Performance measurement under various configurations

Table 2.4 presents the percent improvement from Greedy Utilization as following:

Row 1: One way to look at the results would be with some kind of resource fragmentation score (RFS). We can define resource fragmentation as a function that returns a high score when all resource dimension utilizations have close to equal magnitude, and a low score when the resource dimension utilization magnitudes vary greatly between each other. We can see why a high RFS would be desired if we look at the extremes. the lowest RFS score could be where CPU is at 100% utilization and Memory and Disk are close to zero. In this situation a majority of the machine is not able to be used because the CPU is at maximum capacity and is blocking more pods from being able to be scheduled on the machine. In the opposite situation a very high RFS score could mean that all CPU, Memory, and Disk are used at 100% and there are no wasted resources.

The reason Row 1 constantly performs the best for all rewards comes down to a policy that is able to produce a Resource Fragmentation Score that is on average higher then the other policies. Row 6 with CPU(1) is guaranteed to produce a very low FS. While Row 1 gives the highest possibility for a high FS. Because we are constantly changing the Spiky Resource dimension and because there is almost no repeating Spiky Resources between any two Spiky Pod Sets, it allows a model the maximum opportunity to stack Pods in clever ways to ensure a high FS.

Row 2: This is slightly behind row 1 because in this dataset we allow for the chance of repeating Spiky Resources.

Row 3 and 4: Interesting note that these rows are so similar. This is most likely a coincidence, and it just so happens for our given number of resources and machines that the policies tend to produce similar results.

Never selecting the CPU (row 3) results in an episode termination when one of the other four resources are full across all machines. I suspect that in row 4 the CPU resource is filling up (due to it being selected 50% of the time), and it just so happens to produce a set of pods that fills up the machines in a similar time manner when compared to row 3.

We could test this hypothesis by including or removing a machine and see if we get different test results.

Row 6 and 5 - These rows perform the worst across all resources. This is because if we are constantly selecting the CPU resource to be spiked, this will lead to the cluster quickly filling up the CPU across all machines and leaving the other resource dimensions close to empty. The reason why the model is not able to perform better than the Greedy Utilization is because there are little to no opportunities for improvement. If we can use an analogy from the game "Tetris" this would be like removing the ability to rotate a piece, you can move the Tetris pieces left and right, but without the ability to alter the rotation of the piece, there are only so many ways you can place the piece, and this handicap will result in a game over quickly. The analogy doesn't fit 100% because our trained models cannot "change" a Spiky Resource Dimension of a Pod request like we can with rotating a Tetris piece, but the generalization still stands.

Considering the Resource Fragmentation score then CPU(1) will quickly fill up all machines (leaving all other resources unable to be used) and produce a low fragmentation score.

Utilization Fraction, Min Machine, and Machine Job Fraction Policies: These policies performed not as well because they were trained to reduce energy performance.

Simple constant Policy: This policy performs the best. Most likely the reason it performs better is because we are simplifying the problem. Perhaps calculating the Policy Frag Average Reward ends up complicating things for the model during training. Sometimes the model is forced to make a decision which produces a low fragmentation score, i.e. placing a Pod on an empty machine.

Regarding the CPU(1) score, I believe that if we had 500 to 1k Test instances, the Percent improvement from Greedy Utilization would be much closer to zero. Later we can test this.

2.5.3. Optimization Objective: Energy savings

Energy saving is measured by an indirect indictor, the number of idle machines. We test a burst workload scenario in that a high number of short-lasting pods are submitted to the cluster.

Parameters:

- SPS length 40
- 15 machines
- 5 resource dimensions
- Learning Rate 1e-2
- Test Episodes: 500
- Target update interval (when to update the second model)
    - Every 1000 steps
- Pod Length Mu
    - Training: 150
    - Testing: 40
- Pod Length Sigma
    - Training: 100
    - Testing: 10

Table 2.5 shows the percent of fewer machines used from Greedy Utilization.

Episode termination: The episode terminates after X number of pods have been submitted. During testing we used 5k pods.

Improvement Table: Positive percentage means an improvement, while a negative percentage means Greedy Utilization performs better. Note we are only considering after the machines have been semi-filled. Only considering the last 100 jobs submitted.

Percent of fewer machines used from Greedy Utilization (last 1k jobs)								
Row ID	Random Selection Process	SPS Resource Selection Distribution	Policy Simple constant	Policy Frag Average Reward	Policy Negative Machine	Policy Utilization Fraction	Policy Min Machine	Policy Machine Job Fraction
1	With Removal	Not a uniform distribution	-71.10	-14.13	2.82	5.48	-1.72	-4.77
2	With replacement	Uniform	-99.95	-12.25	2.06	0.29	-6.97	-5.33
3		CPU(0)	-89.60	-7.05	0.89	0.42	-6.04	-4.10

Table 2.5: Percent of fewer machines used from Greedy Utilization (higher is better)

We cannot test rows CPU(0.5), CPU(0.8), and CPU(1) because the machines will almost instantly fill up. The reason we are able to test with CPU(0) is because it allows for a random selection of the four other resources. These four other resources provide enough varied Resource Spiking that the machines don't fill up. The reason we don't' want the machines to fill us is because we are testing for a situation of "moderate cluster fullness". If only a single machine is being used, there is no room for improvement. Inversely if all machines are filled, there is often no room for improvement.

Policy Utilization Fraction: This is the best policy that is able to outperform the Greedy Utilization. While the policies Min Machine and Machine Job Fraction are closer to Greedy Utilization, the policies trained for throughput (Simple Constant and Frag Average Reward) performed much poorer. This policy is able to perform well because it is able to stack the pods in a more efficient manner allowing for a higher Fragmentation score. If you have a higher Fragmentation score, then you will have more pods on less machines when compared to a lower Fragmentation score. The Throughput optimized policy “Simple constant” is also trying to reduce the Fragmentation score but has no constraints with the number of machines it uses, so it tries to use up as many machines as possible.

3. Improved scalability, predictability and stability for edge services

3.1. Forecasting Functional Block architecture

The Forecasting Functional Block (FFB) is a functional block devoted to the computation of forecasting values for given metric(s) of a Network Service. It behaves like a probe, which, consuming the current data related to the selected metric(s), provides a forecasted data stream according to the used model.

The FFB has been designed to interact with other components of the BRAINE architecture, i.e., Distributed Knowledge-base system. It has been implemented in Python, relying on different standard libraries.

In general, the FFB interacts with other BRAINE modules by primarily using the API:

1. REST Server:
    i. receiving as input all the data required, to activate a forecasting job. The input data includes: the input/output Kafka topics, the model's name to be used and the message keys to be mapped over the model default features.
    ii. receiving new trained models, along with details about their default input/output features.
    iii. producing information about existing models, their descriptions and the active forecasting jobs.
    iv. stopping forecasting jobs.
2. Kafka producer/consumer: receiving and producing data from/to Kafka topics according to the running forecasting jobs. The Kafka cluster belongs to the BRAINE Distributed Knowledge-base system.

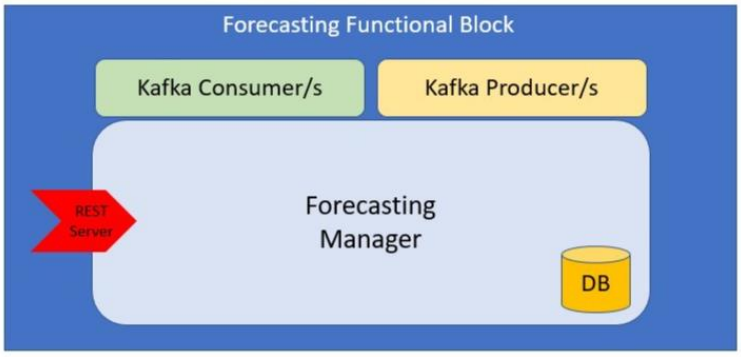

Figure 3.1: FFB High Level Software Architecture

The FFB software architecture is shown in Figure 3.1 and it is composed by the following main blocks:

- **Forecasting Manager**: this block is the core of the FFB. It consists in a python-based project, that includes different modules with different roles:
    - The basis of the module is based on a Flask REST server that implements the APIs to other functional blocks. The details of the

exposed/implemented APIs are detailed in Section 1.1. In general, the APIs allow to both start and stop a forecasting job, according to the input parameters, and to retrieve information regarding the active jobs.

- A part of the module has been defined in define forecasting jobs. It is based on Keras Tensoflow library, where multiple forecasting algorithms can be utilized. Each forecasting job receives data from an instance of Kafka Consumer, able to retrieve current data related to the performance metric to be forecasted, streamed on a dedicated Kafka topic. Each forecasting job runs a specific model. In general, the model must be trained and uploaded in order to be used for forecasting purposes. The FFB can exploit forecasting algorithms based on classical time series analysis (i.e., double exponential smoothing, DES, and triple exponential smoothing, TES) and AI/ML-based.
- The output of the forecasting job is then sent to a dedicated Kafka topic using an instance of Kafka producer, which is filled up with the forecasted metrics.
- The overall state of the module is maintained in a dedicated database. In particular, all the details of the instantiated forecasting jobs are kept up to date, including the job_id, the input parameters, used for the forecasting job activation, the ad-hoc input/output Kafka topic, the selected model name and the downloaded model file.

An example of workflow is described below:

1. A model is pretrained and optimized using offline data collected for a specific aim.
2. The model is then uploaded using the FFB using a specific endpoint available at its REST server. Along with it, a Json object is uploaded containing the default input/output features of the model and number of backward/forward metrics needed/generated by the model.
3. Once a new forecasting job has to be activated, the specific REST endpoint has to be used. The information to be provided are: the input and output Kafka topics, the model to be used, and the input/output features name to be mapped over the default ones.
4. After that, the FFB setups the DB according to the metrics to be stored and starts collecting the input features from the input topic.
5. As soon as the number of collected metrics reaches the *backward metrics* parameter of that model, it starts producing forecasted metrics (the output features) into the output Kafka topic defined.

3.2. Exposed Interfaces

Figure 3.2 shows a swagger view of the implemented API at the Forecasting Functional Block. Following the details of the interfaces are shown.

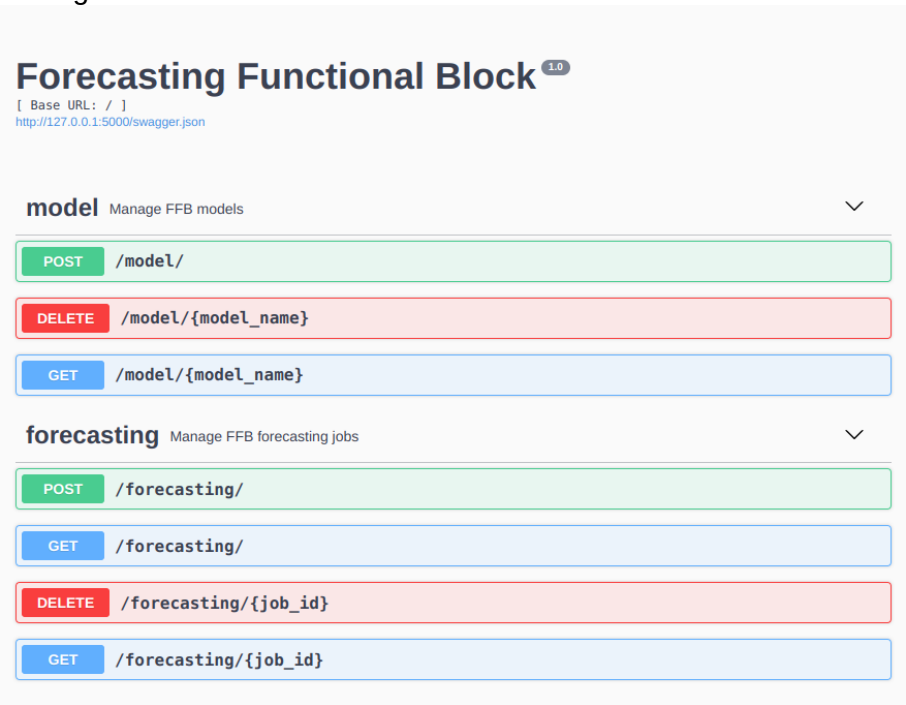

Figure 3.2: FFB API

Forecasting Manager. The APIs exposed by this module enable the creation and the deletion of forecasting jobs. All the requests and responses follow the JSON format. The implemented methods are reported in Table 3.1.

POST /forecasting/	Activate a new forecasting job. Input parameters: • intopic (string) • outtopic (string) • modelname (string) • inputfeatures (dict strings) • outputfeatures (dict strings) Responses: • 200: The job_id is returned. • 404: Forecasting job not started. • 400: Bad request
GET /forecasting/	Retrieve the list of active forecasting jobs. Responses: • *200: The list of the forecasting jobs ("job_id") is returned.*
GET /forecasting/{job_id}	Retrieve the forecasting job details with the specified "job_id". Responses: • 200: The details related to the forecasting job with "job_id" are returned. • 404: Forecasting job not found.

DELETE /forecasting/{job_id}	Stop the forecasting job identified with the job_id: • 200**:** The forecasting job with "*job_id*" is stopped. • 404: Forecasting job not found.

POST /model/	Upload a new trained model.
	Input parameters:
	• modelname (string)
	• modeldescriptor (JSON):
	  • modeltype (string)
	  • definputfeatures (string[])
	  • defoutputfeatures (string[])
	  • backmetrics (int)
	  • fwdmetrics (int)
	• description (string)
	• modelfile (file h5)
	Responses:
	• 200: The model is uploaded.
	• 400: Bad request

GET /model/{model_name}	Retrieve the model descriptor and the description of the model with the specified "model_name". Responses: • 200: The details related to the forecasting job with "model_name" are returned. • 404: Model not found.
DELETE /model/{model_name}	Deletes the model identified with the "model_name" and stops any related jobs. Responses: • 200**:** The model with the specified "model_name" is removed. • 404: Model not found.

Table 3.1: FFB Forecasting Manager APIs

3.3. Usage Example

This section shows an example of usage of the FFB.
In this case, the FFB has been exploited to forecast data from the Mosaic 5G FlexRAN framework managing a 5G testbed.
More precisely, the component consumes SD-RAN metrics from the dedicated Kafka topic, filled by the 3.4 Telegraf agent.
The main feature to be forecasted was the Wide Band Channel Quality Indicator (WBCQI) in download for each UE.
A model has been developed with the following configuration:

- One convolutional layer with 128 filters, kernel size 9;
- Two fully connected layers, with 64 filters and 1 output

It takes in input the t-100 samples in the past, and forecasts t+4. The training data has been collected from the 5G testbed using Telegraf, while manually interfering with the channel quality. The training dataset reaches 30000 samples.

After that the model has been trained, it has been uploaded onto the Forecasting Functional Block by using ForecastingManager API.

The Figure 3.3 shows a plot comparing values for t (in green) from the input topic, the forecasted t+4 (in blue) from the output topic and the actual t+4 (in orange) of the WBCQI value. As you can see, the plot shows the forecasted metric against the actual ones while artificially degradating the signal.

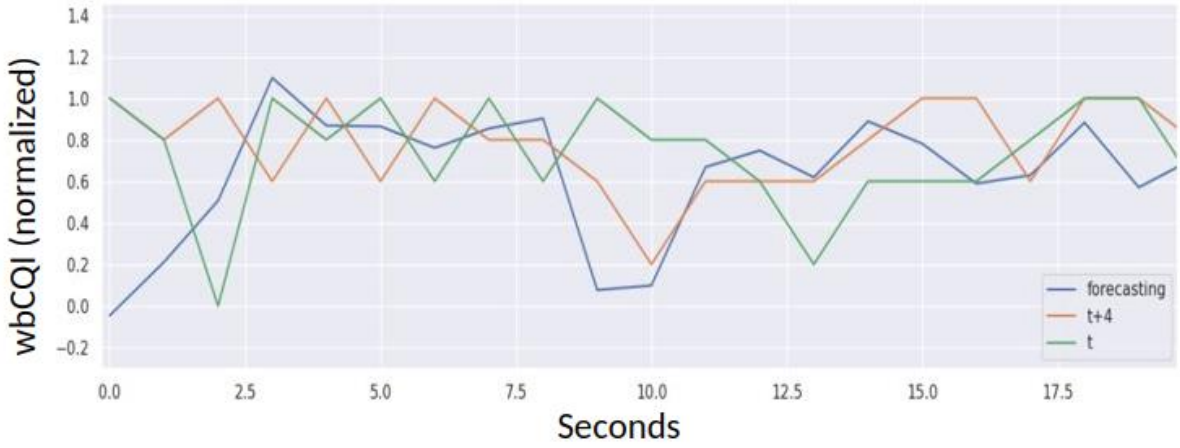

Figure 3.3: FFB Performance in 5G use case

By using the Mean Squared Logarithmic Error (MSLE) among the actual and the forecasted value, the model reached an MSLE of 0.074. This can be further improved by using a larger dataset.

3.4. FFB Conclusions

The developed Forecasting Functional Block (FFB) achieves large flexibility in terms of models and metrics to be forecasted. Thanks to the defined API, new forecasting jobs can be deployed in any moment as far as a model is available. Moreover, this also allows to practically test new versions of a model by executing multiple jobs at the same time. A component with such flexibility at the edge can offer a rapid deployment of new services and functionalities.

4. SDN controller (CNIT)

The standard K8s platform uses a flat network structure that enables Pods to communicate with each other on their hosting cluster. Such flat K8s network, also called Pod network, does not account for network constraints in terms of limited bandwidth or bounded latency. For this reason, deploying K8s in edge computing environments to serve latency/QoS-critical applications requires a specifically designed and comprehensive framework. In particular, specific workflows are needed to efficiently interface K8s with various components such as SDN network controller, Service Level Agreement (SLA) broker, and Telemetry Collector.

4.1. Telemetry workflow

This section presents a comprehensive framework enabling the SDN controller to interact with the K8s scheduler for enabling communication among pods and services deployed on different servers considering the traffic encapsulation applied by the networking tools typically adopted by K8s, such as the “flannel” framework operating in the VXLAN mode. The proposed closed-loop also includes a telemetry system enabling effective SLA monitoring and enforcement. Finally, the framework encompasses an SLA broker interfaced with the telemetry system triggering the SDN controller to perform automated network adaptation upon detection of network performance degradation.

Figure 4.1, illustrates the demonstration workflow as presented in the OFC conference demo Zone in March 2022. The demonstration has been realized using the CNIT cluster deploying a set of pods on two different servers that are interconnected through a packet/optical network using P4-based switches and optical nodes.

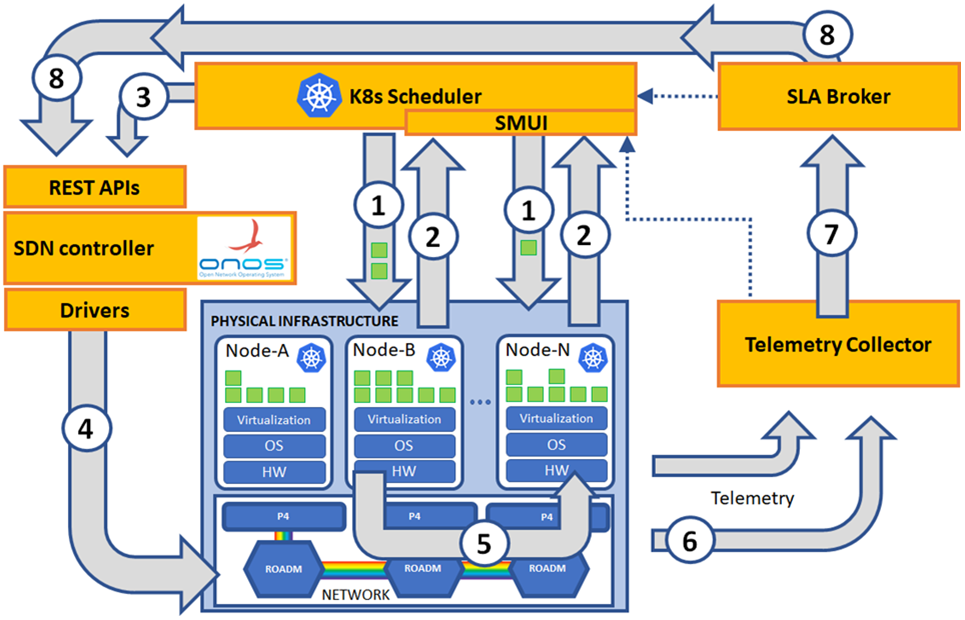

Figure 4.1: BRAINE EMDC main components and closed-loop telemetry workflow.

The demonstration workflow realizes the following steps: Step 1: The scheduler places the Pods with their own requirements on different nodes/locations on the cluster. Step 2: The K8s scheduler retrieves the network parameters of the deployed Pods. Step 3: the K8s scheduler submits a connectivity request to the extended SDN controller feeding the specifically designed ONOS REST interface with the network parameters of the deployed Pods. Step 4: The SDN controller initiates the configuration of the connectivity including both the packet network based on P4 equipment (using P4-Runtime protocol) and the disaggregated metro optical network based on OpenConfig and OpenROADM

yang models (using Netconf protocol), in the same step the SDN controller activates the post-card telemetry on the traversed P4 switches. The telemetry could be also started/stopped in a subsequent step. Step 5: Once the connectivity is configured the traffic starts to flow into the network. Step 6: The related postcard telemetry is generated toward the Telemetry Collector. Step 7: When the SLA broker, that is continuously monitoring the telemetry database, detects a service level degradation (e.g., increased packet loss or latency) it triggers a service upgrade request to the SDN controller using a dedicated method of the designed REST APIs. Step 8: In turn, the SDN controller modifies the network connectivity parameters in accordance with the received request (e.g., find an alternative path on the network).

4.2. SDN controller applications

The BRAINE SDN network controller is based on ONOS [ONOS]. Figure 4.2 represents the components specifically developed for BRAINE and utilized in this work to implement traffic forwarding and in-band telemetry, i.e., the BRAINE app and the BRAINE P4 app.

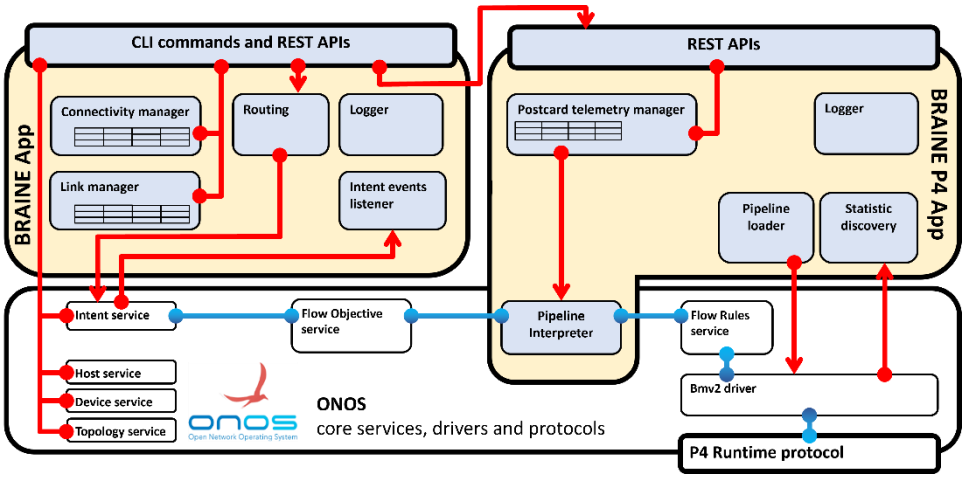

Figure 4.2: Internal architecture of the ONOS apps developed for the BRAINE project, including relations with ONOS core services, drivers and protocols. Red connectors represent relations implemented within this work, blue connectors represent relations already present in the ONOS core.

4.2.1. The BRAINE app

This application implements a set of functionalities exposed through REST APIs, enabling the interaction with Kubernetes, and the SMUI. Also, the same functionalities can be manually accessed through a set of CLI commands. Moreover, the application utilizes the ONOS core services to enable the deployment of point-to-point connections between pods running in different worker nodes of the cluster. The two main functionalities supported at the data plane by the BRAINE app are: i) connection management (i.e., add/delete/modify), where each created connection can be specified up to the transport level (i.e., TCP/UDP ports); ii) activation of telemetry on selected active connection(s).

To support the aforementioned features, the BRAINE app is composed of several components (see left side of). In particular the application includes: i) two databases where connection and link state information are stored; ii) a routing module that performs redundant routing of requested connections and interacts with the ONOS intent service; iii) an intent listener that allows the application to react in case of network events affecting established connections; iv) a logger for tracing and debugging. Moreover, the BRAINE app supports a set of accessories features to facilitate the interaction with the network and the gathering of network state information. Specifically, the features

supported by the app can be grouped in four categories: connections related commands, device related commands, host related commands and link related commands.

4.2.2. The BRAINE P4 app

The companion BRAINE P4 application has been developed to program the specific P4 pipeline to be used in the data plane switches. This application has two main roles: i) enabling the match of header field encapsulated within VXLAN tunnels; ii) activating the postcard telemetry on specific traffic flows. The first objective is achieved through the implementation of a dedicated pipeline (described in next section). For the latter objective, the application exposes a REST API that is dynamically consumed by the BRAINE app when a telemetry activation request is received from the orchestrator.

The internal architecture of the BRAINE P4 application is represented on the right side of Figure 4.2. It includes the *pipeline loader* component which loads the P4 pipeline description via the P4Runtime protocol upon the discovery of P4-based switches. Once the request to activate a new postcard telemetry on a specific traffic is received through the REST interface, the Postcard telemetry manager identifies the devices traversed by the flow and sends them the flow rules to enable the postcard via the pipeline interpreter. Since the pipeline interpreter is the only component that is aware of the pipeline structure (e.g., number of tables and supported matching fields per table) it is also used for translating into flow rules the output of the intent service created to forward traffic. The statistic discovery component collects traffic related information from the P4-based devices to be visualized in the ONOS GUI (e.g., counters associated to flow rules). Finally, the logger component facilitates tracing and debug.

The BRAINE P4 app then relies on the Bmv2 P4 driver included in the master ONOS master distribution that has been demonstrated to be fully functional to perform the connection to P4 devices and to install all the required flow rules using the P4 Runtime protocol.

4.3. P4 pipeline implementation

The developed P4 program is written in P416 for the target architecture v1model [V1Model] that includes a parser and two pipelines (ingress and egress). With the proposed approach the P4 device can be programmed by the SDN controller to forward both traffic exchanged among pods (i.e., encapsulated using VXLAN) and traffic exchanged among host machines (i.e., not encapsulated). Moreover, the controller can activate in-band telemetry (i.e., postcard telemetry, INT-XD) on selected traffic flows, that can be specified up to transport layer details (i.e., TCP/UDP ports).

The proposed architecture is working only in conjunction with the Flannel CNI plugin, that is the one selected to be used in all BRAINE deployments. However, it is easily extensible to other tunneling techniques applied by different CNI plugins, only requiring the upgrade of the parser module.

Each pipeline is composed by a number of tables, operating with a match/action policy. Each table supports a specific set of keys and actions. In each table, a ternary match policy is used where the selected mask allows to ignore a key or apply an exact match. Some keys are packet header fields, while others are custom metadata that are associated to the packet during the parsing procedure.

4.3.1. P4-based matching of pod-to-pod traffic

The parser, detailed in Figure 4.3(a), is the first module of the ingress pipeline, as shown in Figure 4.3(b). While the packet passes through the parser stages, its header fields and the metadata fields are gradually filled. The first stage of the parser writes the ingress port index into the specific metadata field. Then, the *Parse Packet IO* stage is executed only for packets received from the CPU port (i.e., P4 Runtime packet_out messages received from the controller) to retrieve the packet_out header. The *Parse Eth* stage extracts the Ethernet header, moreover it fills the fields local_metadata.l2_src_addr and local_metadata.l2_dst_addr} with the values contained in the MAC source and destination fields. Then, in case of IP packets, the *Parse IPv4* stage is executed parsing the IPv4 header, and filling the metadata fields local_metadata.l3_src_addr and local_metadata.l3_dst_add with the IP source and destination addresses. Subsequently, the packet is sent to one of the *Parse TCP/UDP* stages where the metadata fields local_metadata.l4_src_port and local_metadata.l4_dst_port are filled.

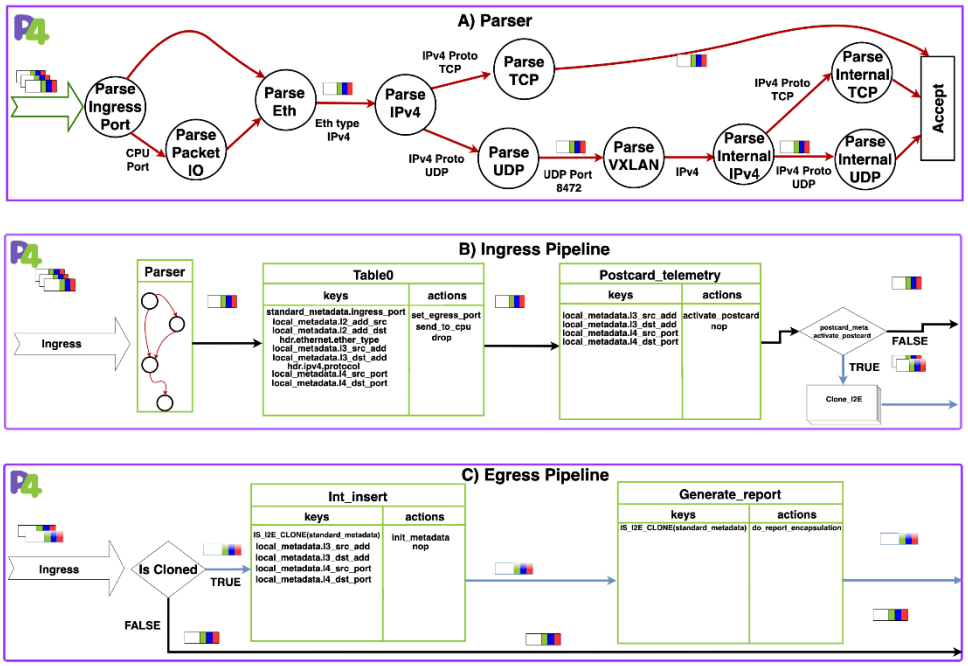

Figure 4.3: Proposed pipeline architecture for traffic forwarding and telemetry: a) Parser; b) Ingress pipeline; c) Egress pipeline.

If the UDP destination port is 8472, it means that the packet belongs to a pod-to-pod traffic flow encapsulated within a VXLAN tunnel by Flannel. In this case, the *Parse VXLAN* stage is executed parsing VXLAN header, subsequently IP and TCP/UDP headers are parsed by *Parse Internal* stages. During these stages, the aforementioned local_metadata.* fields are overwritten with the corresponding fields enclosed in the internal headers. This way, if the packet is encapsulated in a VXLAN tunnel, the ingress pipeline will match the internal header fields, thus enabling pod-to-pod traffic forwarding.

As illustrated in Figure 4.3(b), after parsing, the packets are forwarded to the ingress pipeline and processed by table0 where the egress port is assigned based on the flow rules installed by the SDN controller.

4.3.2. P4-based postcard telemetry implementation

The subsequent tables in both the ingress and the egress pipelines are used to implement the postcard telemetry. The Postcard_Telemetry table, see Figure 4.3(b), matches on metadata fields and is intended to contain flow rules for matching each

traffic flow requiring postcard telemetry. Two actions are supported: *activate_postcard* and *nop*. The action *activate_postcard* is executed for each matching packet (i.e., to packets belonging to traffic flows for which the SDN controller has activated the telemetry), setting a specific metadata field (i.e., meta_activate_postcard) that is later evaluated by an if condition to clone the packet using the *cloneI2E* external feature. If a packet is not matched, the default action *nop* is executed resulting in the packet forwarded to the egress pipeline without cloning. The cloned packet will be manipulated in the egress pipeline to generate a report packet.

The egress pipeline is illustrated in Figure 4.3(c). All the metadata fields local_metadata.* must be re-initialized because P4 does not allow the propagation of custom metadata from the ingress pipeline to the egress pipeline. No actions are applied to the original packet that leaves the switch through the port assigned in table0. Instead, the cloned packet is processed by the two tables: *int_insert* and *generate_report*. The former table, with a null default action, applies the action *init_metadata* to matching packets. This action is the one that actually retrieves the information to be included in the report message that is written in the *local_metadata.postcard_** fields.

The latter table generates the in-band telemetry report message using the action *do_report_encapsulation* manipulating the cloned packet. More in detail, the header of the cloned packet is modified as following. The Ethernet and IP source addresses are set to the local switch values, while the destination addresses are set to the telemetry collector values. The UDP source and destination ports are set to a specific value to easily recognize report packets at the telemetry collector. Finally, the report header is added as UDP payload that includes the metadata retrieved in the previous table, i.e., switch_id, flow_id and all other metadata required by the SDN controller using the instruction_mask as defined in [INT].

4.4. Experimental demonstration results

4.4.1. Experimental setup

The experimental testbed encompasses both computing and networking resources. Computing resources are deployed on two dedicated servers, i.e., EMDC1 and EMDC2 in Figure 4.4. The hardware of both servers is a DELL PowerEdge R740, 56 CPUs Intel Xeon Gold 6238R @ 2.20GHz, 256 GB RAM. Three virtual machines (VMs) are deployed in EMDC1, while two VMs are deployed in EMDC2. One of the VMs deployed on EMDC1 hosts the management and control software including the Kubernetes master, the ONOS SDN controller, the telemetry collector and the telemetry and monitoring platform. The other VMs act as Kubernetes worker nodes, where each node runs a number of pods (i.e., each pod encompasses a plain Ubuntu 20.04 distribution with basic networking tools). The Telemetry and Monitoring platform includes the telemetry database deployed into an influxdB container, and the SLA Broker, implemented as a set of configurable queries and threshold-based alarms through dedicated Grafana panels.

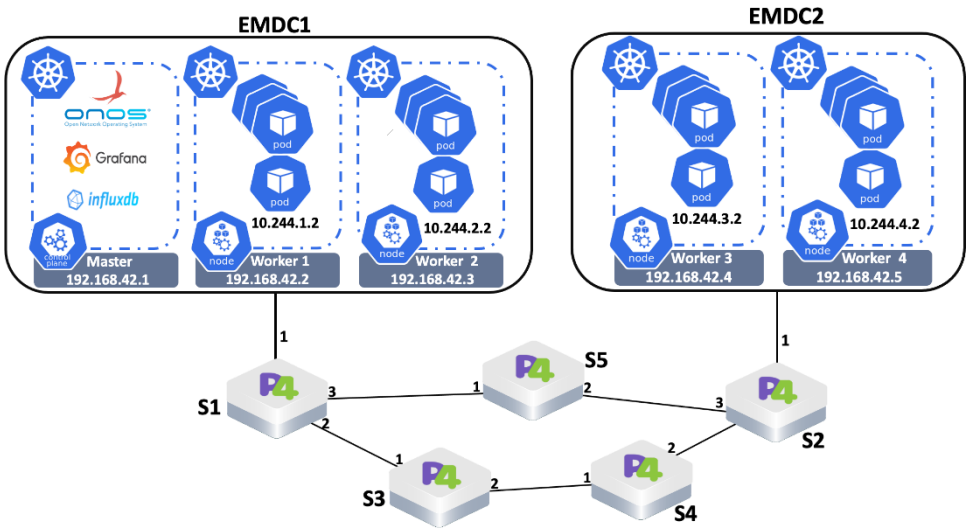

Figure 4.4: Experimental testbed encompassing computational and networking resources.

Networking resources encompass five P4-based switches, all of them emulated using Bmv2. Switches S1, S2, S3, S4 are emulated on a dedicated DELL server (Intel Xeon E5- 2643 v3 6-core 3.40 GHz clock, 32 GB RAM) using physical Ethernet interfaces. Switch S5 is emulated by deploying a dockerized Bmv2 on a Mellanox SN2010, running SONiC. The traffic report generated by the network nodes is received by the Telemetry Collector, hosted by the Kubernetes master node. As depicted in Figure 4.4 the report packet contains: the switch_id field that identifies the switch, the flow_id field that discriminates traffic flows, Ingress_Timestamp and Egress_Timestamp needed to evaluate the hop latency.

4.4.2. Experimental results

1) Pod traffic forwarding validation

This section functionally validates the proposed solution to process the traffic exchanged between a pair of Kubernetes pods. Specifically, the traffic is generated between two pods respectively deployed on node EMDC1 and EMDC2, thus traversing the P4-based network. Figure 4.5 illustrates the Wireshark capture, including the VXLAN encapsulation and the protocol stacking as applied by the Flannel CNI plugin. Specifically, the ping application is used to generate ICMP request/reply messages between pod 10.244.1.2 deployed on worker node 1 and pod 10.244.4.2 deployed on worker node 4. The packets are captured in VM Worker 1 on interface 192.168.42.2. The presence of both ICMP request and reply proves that packets are correctly switched by the network in both directions. The experienced round-trip time is around 5 milliseconds.

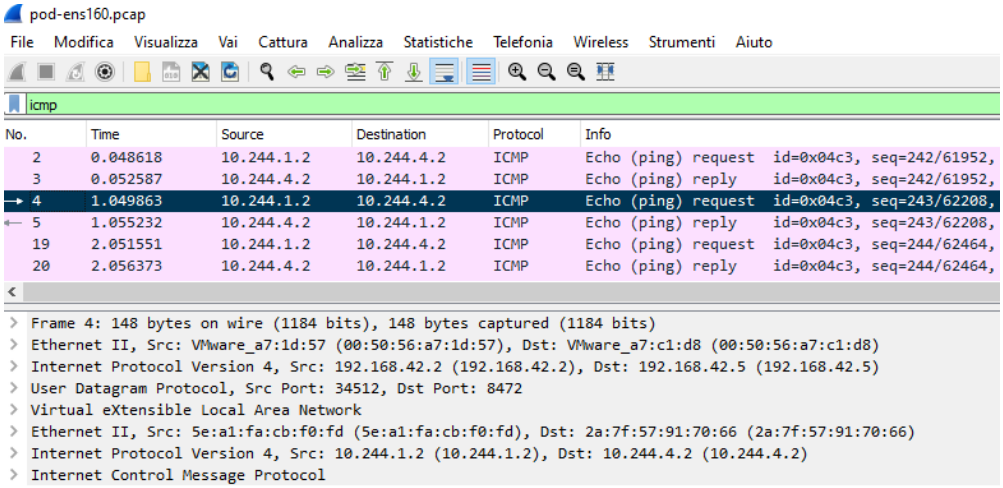

Figure 4.5: Wireshark capture of ICMP traffic between two pods.

Figure 4.6 shows a screenshot of the ONOS web GUI illustrating the flow rules installed in switch S1 where the rules counters show that the traffic exchanged between the two pods is correctly matched.

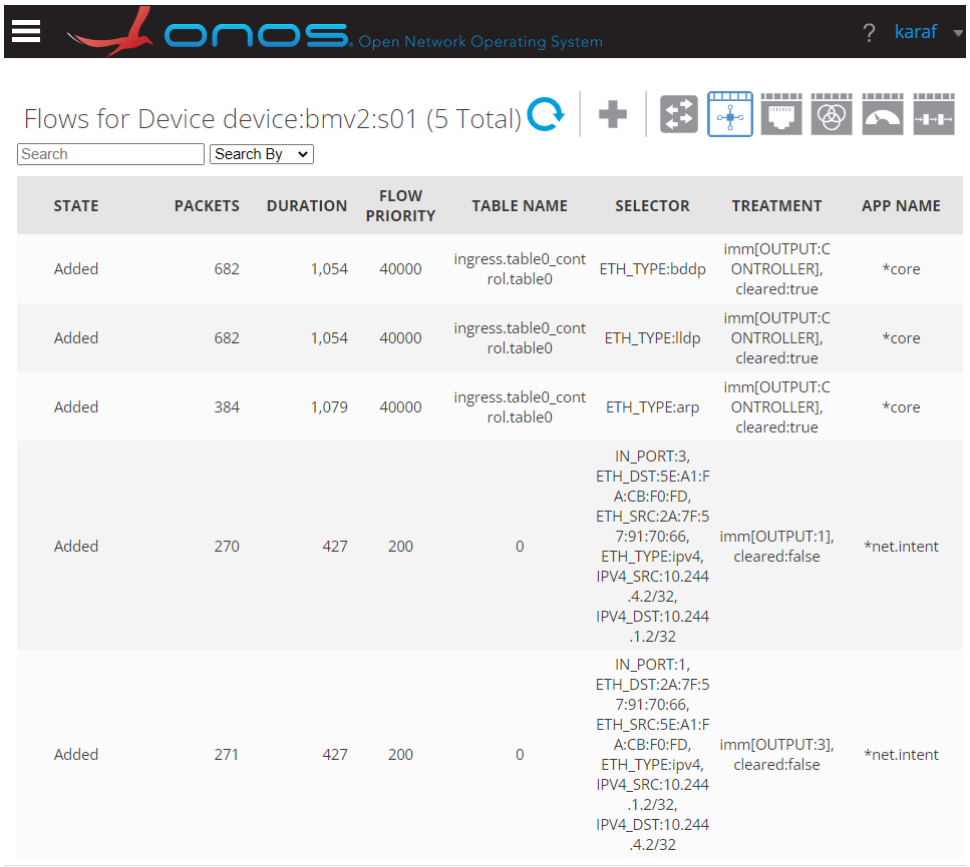

Figure 4.6: ONOS view of rules installed on switch S1.

2) Pod traffic telemetry validation

This section functionally validates the whole telemetry workflow as described in Fig. AAA. Specifically, two separate traffic flows are activated between two different pairs of pods: flow IDs 250 and 123. The two flows consist of five parallel TCP sessions generated with the iperf3 application. Telemetry is active in both flows; however, the SLA

Broker is configured to generate the feedback to ONOS (step 8 in Fig. AAA) only for flow 250.

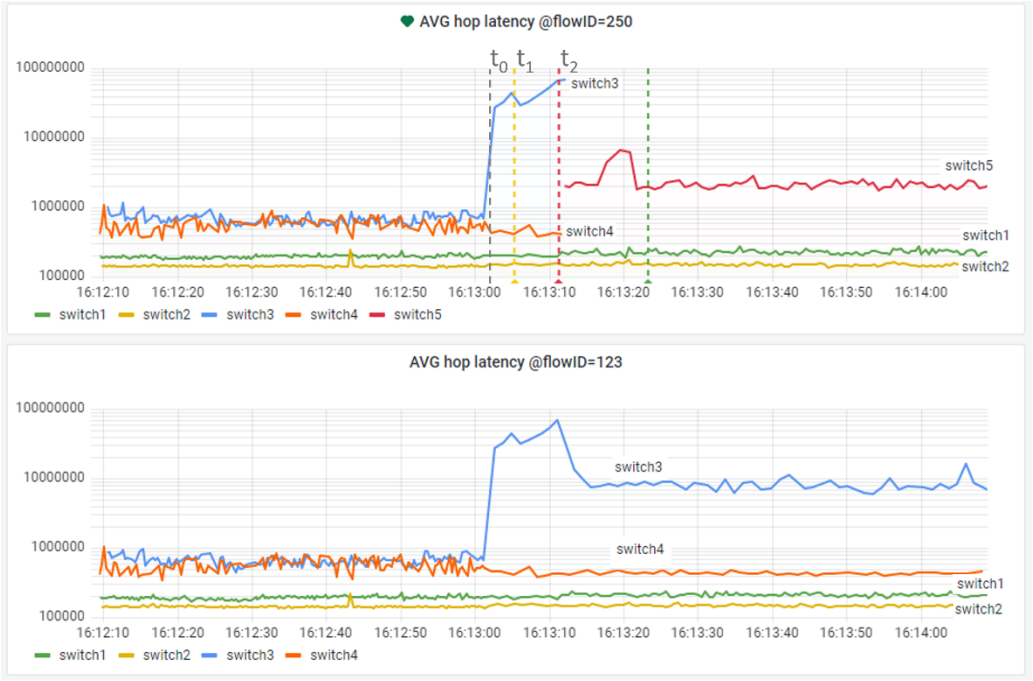

Figure 4.7: Monitoring Platform: view of switch latency for traffic flows 250 and 123. Latency [ns] as a function of experiment time.

Figure 4.7 reports the latency data as collected by the SLA Broker panels during the network reconfiguration. Both flows are initially routed along the path S1, S3, S4, S2, thus both plots report four latency lines, one per traversed switch. At time t0 switch S3 transmission rate is manually degraded, thus increasing the switch latency for both flows. The SLA Broker performs a threshold-based control over the per switch latency of flow 250 and triggers an alert if the degradation persists for 4 seconds. This behavior is reflected in the SLA Broker panel as depicted in Figure 4.7. In the actual experiment, degradation is detected at t1 and the alert is triggered back to ONOS at t2. As described in the previous sections, ONOS reacts by rerouting the affected flow (i.e., flow 250) on path S1, S5, S2, i.e., after t2, Figure 4.7 reports the latency of those switches. It is worth noting that S5 is characterized by a higher latency compared to other switches; indeed, S5 is emulated on less performance hardware. Conversely, flow 123 is not involved in the reconfiguration, showing that the implemented framework is able to select the single traffic flow.

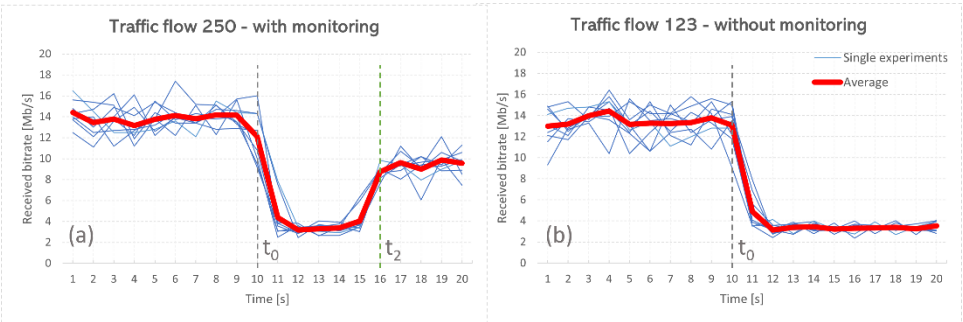

Figure 4.8: Flow bitrate: (a) flow 250; (b) flow 123. Mbps as a function of experiment time.

The telemetry workflow experiment has been repeated 10 times collecting also the achieved end-to-end bit-rate of both flows. The results are illustrated in Figure 4.8,

including ten cyan lines reporting the specific result for each experiment and a single red line reporting the average trend. Specifically, Figure 4.8(a) is related to traffic flow 250, it shows that after t0 the rate is degraded, then it is partially recovered at time t2 when the traffic is switched on the alternate path. It is worth noting that rerouting the traffic does not guarantee the recovery of the overall bit-rate. In fact, the recovery path includes switch S5 emulated on a less performing hardware with limited traffic capabilities. Figure 4.8(b) is related to traffic flow 123 that is not involved in the reconfiguration, thus after t0 the bit-rate results to be degraded and never recovered. Figure 4.8(a) shows that the whole workflow takes about 6 seconds to be performed (i.e., t2 − t0). However, most of this time is expended within the telemetry and monitoring platform (i.e., SLA Broker) as a result of our configuration to trigger the alert.

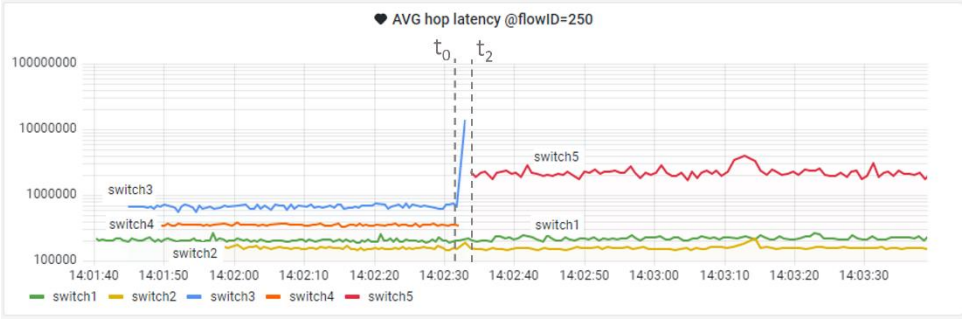

Figure 4.9: Auxiliary panel view of switch latency for traffic flows 250 experiencing network failure recovery excluding the Telemetry and Monitoring Platform. Latency [ns] as a function of experiment time.

This time could be reduced by configuring the SLA Broker with higher SLA checking rates on the InfluxDB filled by the Telemetry Collector. Therefore, to better evaluate the achievable performance of the system, we have measured the re-configuration time excluding the telemetry and monitoring platform from the workflow, i.e., the feedback to the ONOS controller is directly generated by the Telemetry Collector. Figure 4.9 reports the latency data collected by an auxiliary Grafana panel during the network reconfiguration, when the reconfiguration is triggered directly by the Telemetry Collector (i.e., thus excluding the influxdB and the SLA Broker). The experiment has been repeated 10 times and the average time for performing the reconfiguration is 1.95 seconds that includes: the detection of the latency degradation at the Telemetry Collector, all control plane procedures performed in ONOS (e.g., computation of an alternate path), and P4 Runtime message exchange towards the involved switches.

5. DevOps for edge computing supporting AI

This deliverable has presented the main activities in year-3 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.

Over the last interaction, we have extended the vocabulary to support workflow placement and description. The schema was extended to support Images descriptions through Docker deployments descriptor format (Listing 5.2). Services, using Kubernetes deployment descriptors (Listing 5.1) and Workflows using Argo description language which is an extension of Kubernetes deployment description language itself. We call those descriptions manifest. By relying on an attribute called manifest we allow the use of the same model later to other existing description languages. Figure 5.1 gives an overview of the vocabulary developed for resource management and service deployment.

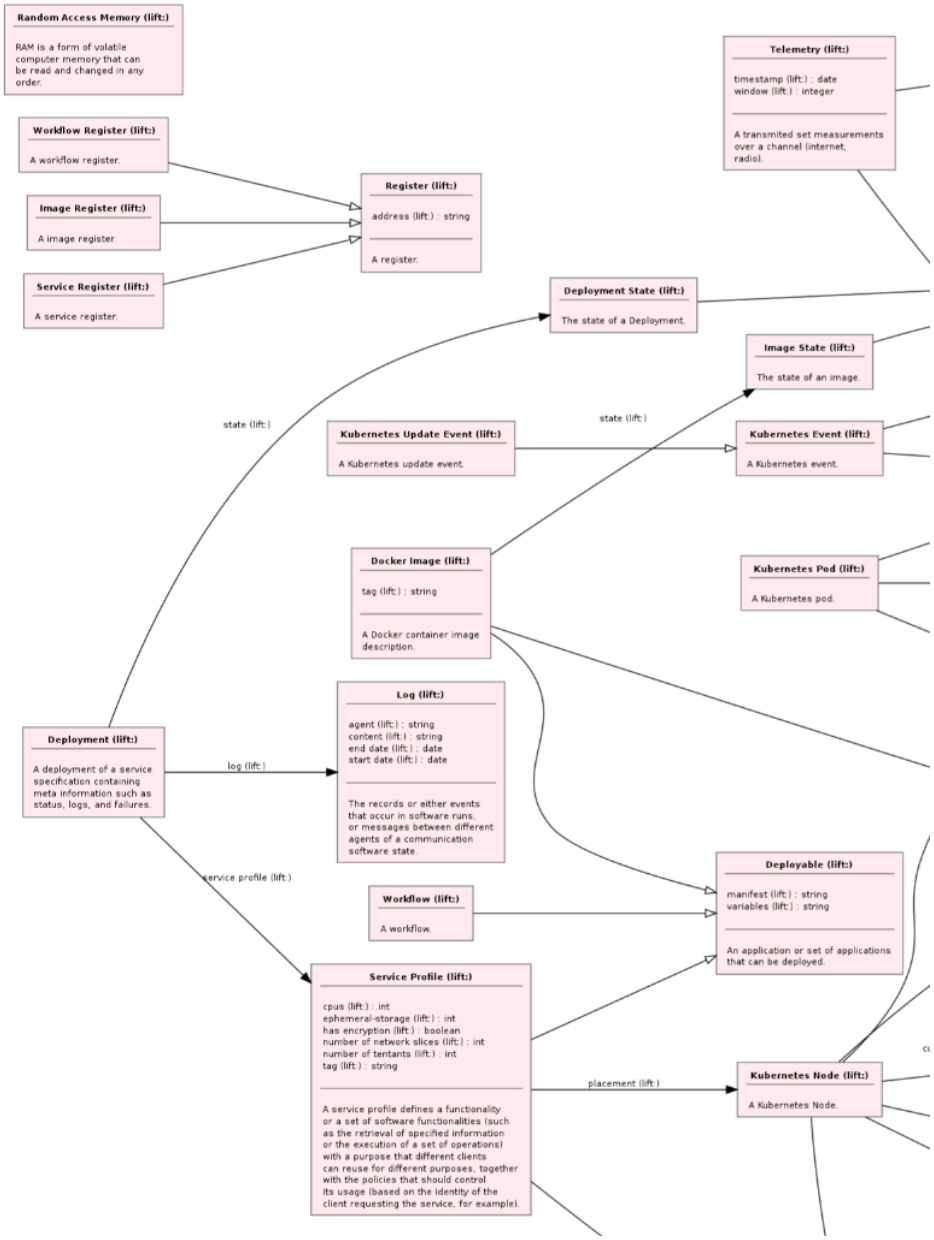

Figure 5.1: Excerpt of BRAINE schema (lift) highlighting Service, Workflow and Image Descriptors.

```
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
```

Listing 5.1: Kubernetes manifest example.

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

Listing 5.2: Image manifest example.

In addition, we have also extended the vocabulary (Figure 5.2) allowing users to add services, images and workflow register endpoints to the BRAINE knowledge graph. This extension allows users to directly interact with the endpoints without the necessity of using the Service or Image Orchestrator. This approach significantly simplifies the deployment and management.

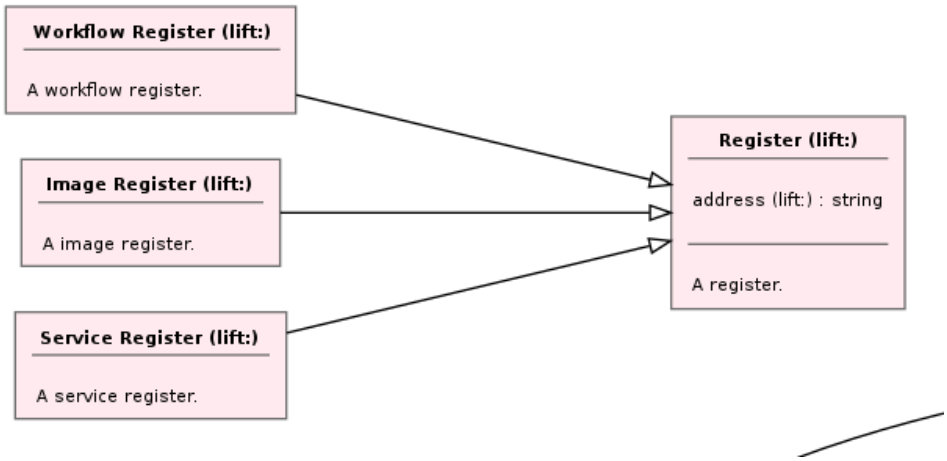

Figure 5.2: Excerpt of BRAINE vocabulary lift highlighting Workflow, Image and Service Registries.

Workflow Definition For being open source and tightly integrated with Kubernetes, Argo was chosen as the service workflow execution framework. In addition, it offers all required functionalities in the project scope and users can describe workflows in a declarative way using manifests in a similar fashion to those of Kubernetes and Docker.

With the addition of the Argo framework to the BRAINE software stack, it is possible to define workflows through Argo workflow definition language (Listing 5.3). Argo can be easily coupled with the overall project architecture and can be easily managed by the user as well as by the system. Argo's tight integration with Kubernetes, its declarative workflow definitions, and its support for event processing make it the most suitable candidate. It is likely that effort will be required to integrate Argo into the BRAINE architecture and develop its functionality further; but this is preferable to building a

custom BRAINE solution from scratch, and may provide useful contributions to the opensource solution. The full vocabulary is available under Creative Common CC-BY-4.0 license at https://github.com/eccenca/braine-vocab.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: hello-world-
  labels:
    workflows.argoproj.io/archive-strategy: "false"
  annotations:
    workflows.argoproj.io/description: |
      This is a simple hello world example.
      You can also run it in Python: https://couler-proj.github.io/couler/examples/#hello-world
spec:
  entrypoint: whalesay
  templates:
  - name: whalesay
    container:
      image: docker/whalesay:latest
      command: [cowsay]
      args: ["hello world"]
```

Listing 5.3: Example of workflow definition.

Registry Interfaces: In addition to the schema extension, we have also further developed a BRAINE management webclient that allows the management of services, images, workflows and their respective registries (Figure 5.3). Figure 5.4 displays the Docker Image register window in CMEM, it allows users to register Docker images for deployments. Figure 5.5 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 in case of Service Profile and Docker Image Descriptor in case of Docker Images.

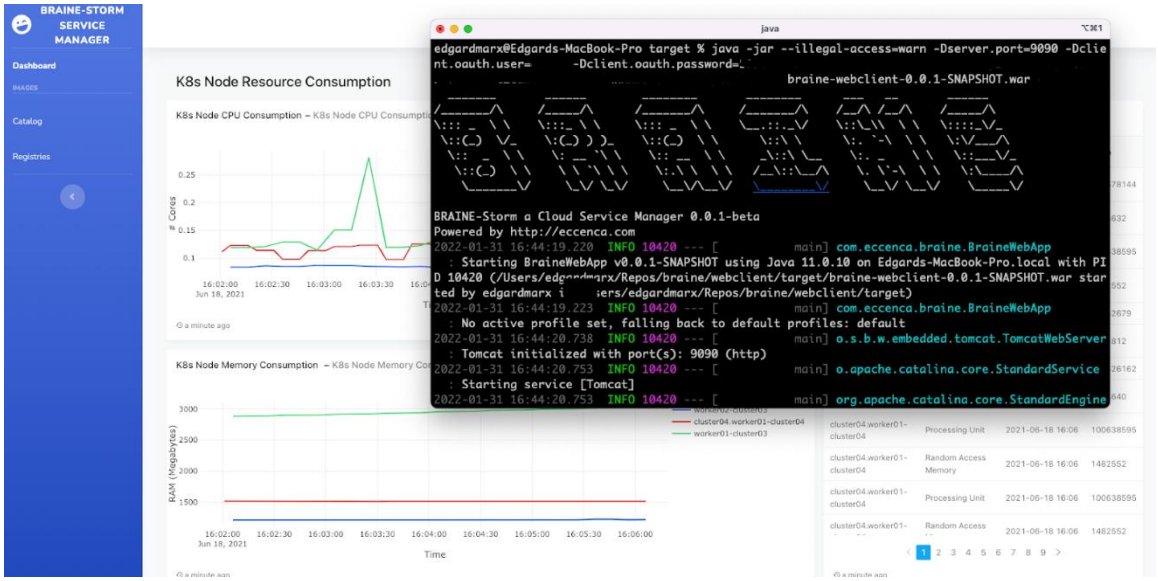

Figure 5.3: BRAINE webclient.

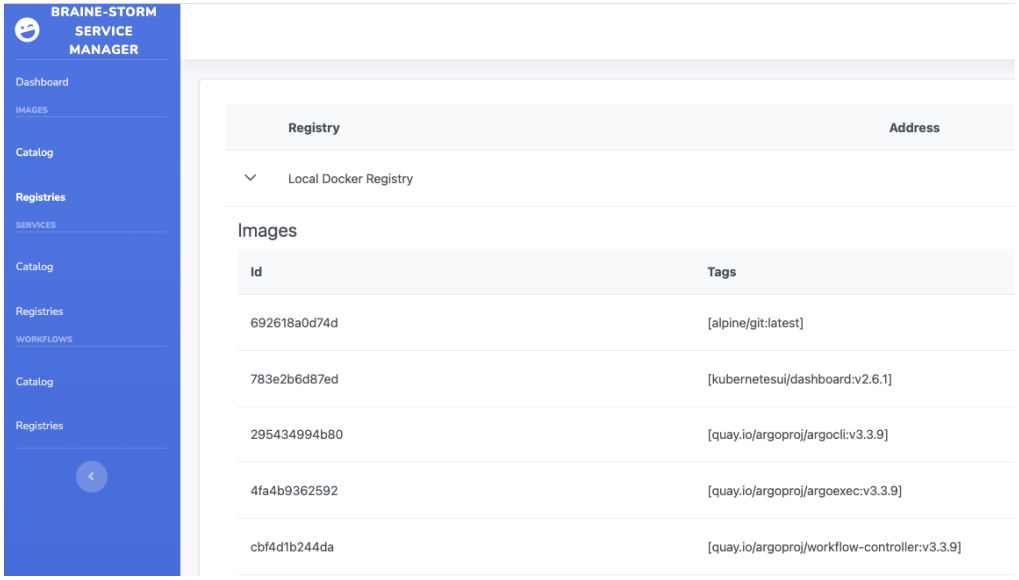

Figure 5.4: BRAINE webclient Image Registry Web Interface.

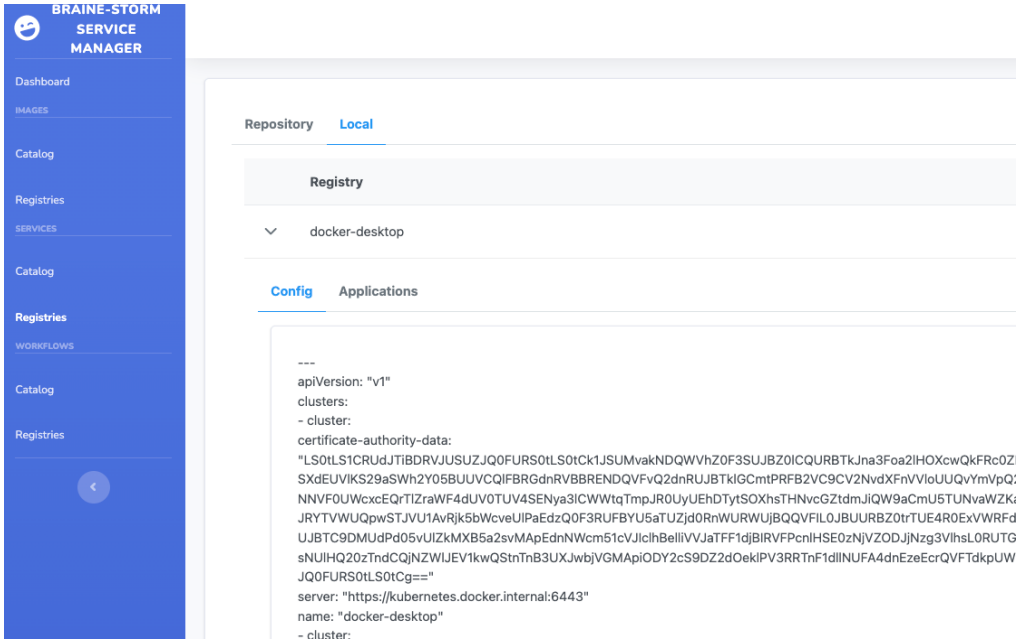

Figure 5.5: BRAINE webclient Image Registry Web Interface.

6. Monitoring and SLA broker incorporation for transparency and enforcement

To resolve system violations, the SLA Broker plays a crucial role. This section provides an overview of the SLA Broker requirements for the BRAINE system, describes the components that make up the BRAINE SLA Broker system, and demonstrates the workflows involved in the process.

6.1. System requirements

The design and development of the BRAINE SLA Broker considered the following system requirements:

- **Policy-based system**: A single-metric system (e.g., bandwidth) for failure detection and handling may not be sufficient to properly resolve a failure. Context is essential for system diagnosis, as different contexts may require distinct solutions. A policy-based system enables the capture of a system violation context based on several metrics and conditions. A BRAINE SLA Broker policy consists of one or more rules, with each rule monitoring the violation of a metric performance over a defined period. This includes the type of metric aggregation, the number of violations within a defined period of time, the trigger type, and the violation threshold. The violation type can be either one metric violation to trigger the actuation or a combination of metric violations to trigger the actuation. Table 6.1 illustrates the fields of a rule, which can be generated using a POST REST API request to "/rule/". Note that a rule is only associated with one policy.

Field	Type	Description
ID	UUID	A **returned value** for a successful POST request. The ID field can be used later to retrieve, update, or delete the rule. This value is system generated and will be ignored if submitted for a new rule.
Name	String	Name of the rule.
OwnerID	UUID	Owner ID should match the owner ID of the associated policy
Description	String	A description for the rule.
Parameter	Dictionary	Consists of the "endpoint", type of measurement provider (i.e., Prometheus, InfulxDB, RESTapi, or message bus), field name.
Operator	String	It is one of the following operators ">", "<", ">=", or "<=". This field is used as indication to interpret the threshold.
Period	Integer	Period is in seconds. It represents the considered window to apply the rule. For instance, Period equals 600s means the window that the SLA rule is applied on is the last 600 seconds (i.e. 10 mins).
Occurrence	Integer	The number of violations in the period to

		trigger a rule violation.
Interval	Integer	Interval is in seconds. It is the time interval to pull the metric read/reads from an endpoint.
Threshold	Double	The threshold value
TriggerType	Boolean	“0” or critical means that the actuation can be trigger only by violating this rule. “1” or non-critical means that a number of this type of rule (more than one should be defined in the policy) need to be violated to trigger an actuation.
PolicyID	UUID	The policy that a rule is associated with.

Table 6.1: Fields of the SLA rule

Table 6.2 illustrates the fields of an SLA policy entry, which can be generated using POST RESTapi request to “/sla-broker/”.

Field	Type	Description
ID	UUID	A **returned value** for a successful POST request. The ID field can be used later to retrieve, update, or delete the policy. This value is system generated and will be ignored if submitted for a new policy.
Name	String	Name of the policy.
OwnerID	UUID	Owner ID
Description	String	A description for the policy.
RuleThreshold	Integer	The number of rule type “1” that required to trigger an actuation.

Table 6.2: Fields of the SLA policy

Table 6.3 illustrates the fields of an actuation entry, which can be generated using POST RESTapi request to “/actuation /”.

Field	Type	Description
ID	UUID	A **returned value** for a successful POST request. The ID field can be used later to retrieve, update, or delete the actuation entry. This value is system generated and will be ignored if submitted for a new actuation entry.
Name	String	Name of the policy.
OwnerID	UUID	Owner ID
Description	String	A description for the policy.
Endpoint	Dictionary	It consists of the URL, communication type (i.e., RESTapi or KAFKA), KAFKA

		topic.
Message	Dictionary/String/Value	Dictionary lists the UUID of violated rules. String and value are fixed and provided by owner.

Table 6.3: Fields of the SLA actuation

- **Metric endpoint**: Metric values can come from several sources, including the producer itself and aggregation points of a telemetry system. The BRAINE Telemetry system provides several methods for retrieving telemetry values using PIGPI, where PromQL and InfluxQL can be used to retrieve the measurements. In some cases, the BRAINE Telemetry system also allows for telemetry retrieval from the KAFKA bus.
- **Metrics analysis**: Telemetry data needs to be aggregated and analysed over a period to produce output that can be relied upon to trigger actuations. The analysis may consider patterns, a single metric, or multiple metrics.
- **Actuation approaches**: Violation of an SLA Broker policy results in one or more actuations to handle the SLA policy violation. The actuation can be as simple as sending an email to the owner or administrator about the violation. Additionally, the actuation can be system-accommodated, such as scaling in or scaling out the microservice deployment. Finally, the actuation can be application-specific, which requires the microservice owner to provide the customized message format and content to the SLA Broker.

6.2. System Components

6.2.1. SLA Broker Manager

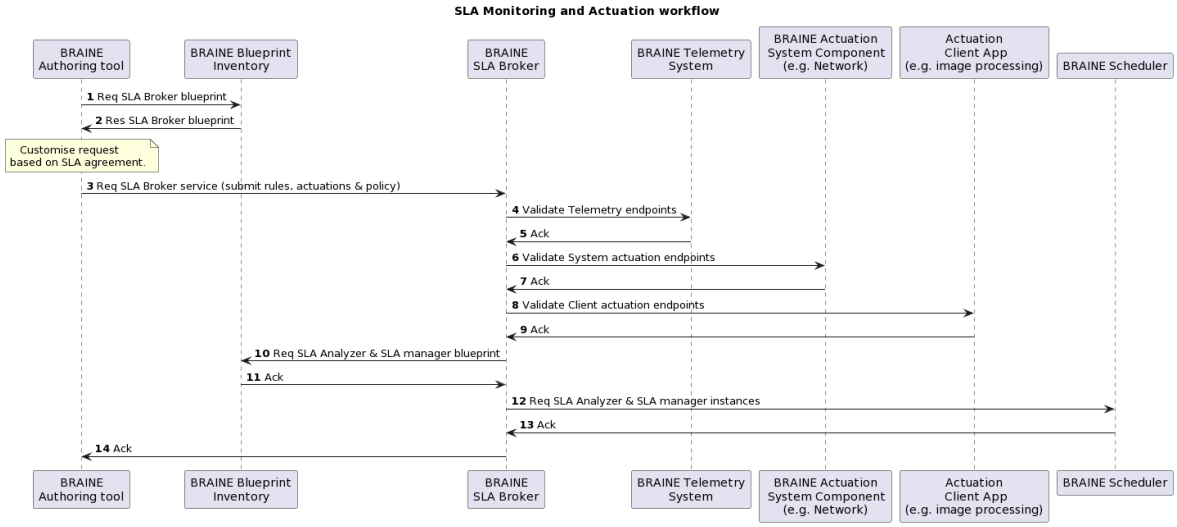

Figure 6.1: the workflow for successfully instantiating SLA Analyzer and Manager instances

Figure 6.1 illustrates the workflow for successfully instantiating SLA Analyzer and Manager instances. Firstly, the BRAINE Authoring Tool requests the blueprint for creating SLA rules, actuations, and policies from the BRAINE Blueprint Inventory (steps 1-2). Secondly, the BRAINE Authoring Tool customizes the SLA policy request based on the SLA agreement and submits it to the BRAINE SLA Broker (step 3). Thirdly, the BRAINE SLA Broker validates the telemetry and actuation endpoints (steps 4-9). Fourthly, the SLA Broker requests the blueprints for the SLA Analyzer and Manager from

the BRAINE Blueprint Inventory (steps 10-11). Then, the SLA Broker customizes the Analyzer instance to collect the targeted telemetry from the proper endpoints. The customization of the Analyzer instance also includes customizing the communication between the SLA Analyzer and Manager instances. Later, the SLA Broker requests the related deployment and services from the BRAINE scheduler (steps 12-13). Finally, the BRAINE SLA Broker confirms the deployment of the SLA Broker related instances to the BRAINE Authoring Tool.

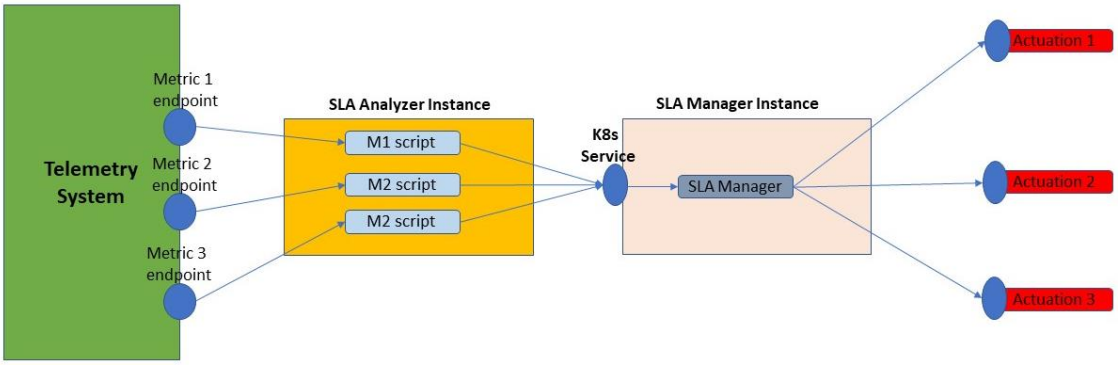

Figure 6.2: An example of an SLA Broker deployment

Figure 6.2 shows an example of an SLA Broker deployment. In the figure, the SLA Analyzer consists of three metric scripts that pull telemetry readings from the Telemetry System based on the interval fields in each rule submitted to this policy. In case of a rule violation, the related script sends a notification message to the SLA Manager about the violation. Based on the type and number of violated rules, the SLA Manager instance communicates with the actuations.

6.2.2. SLA Analyzer instance

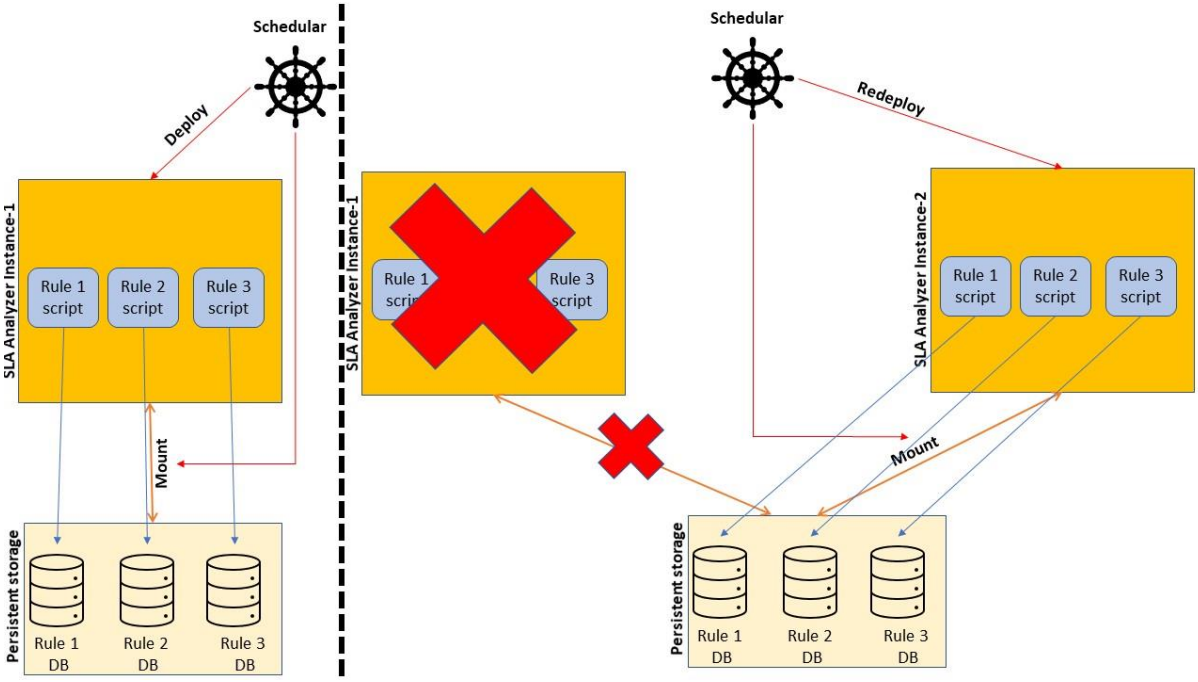

Figure 6.3: SLA Analyzer deployment

Figure 6.3 illustrates the deployment of Analyzer instances. As shown in the figure, Analyzer instances are associated with persistent storage to ensure system monitoring in case of Analyzer failure or crash. In the event of Analyzer failure, the scheduler instantiates another Analyzer and mounts it to the persistent storage. An Analyzer instance keeps a copy of all measurements and related analyses on the persistent storage. Moreover, the figure shows that each rule is monitored by one script and that the logs and readings are stored on the persistent storage.

6.2.3. SLA Manager Instance

The SLA Manager Instance is responsible for calling actuations based on the violations of the rules reported by the SLA Analyzer. As mentioned earlier there are two types of reactions for rule violation.

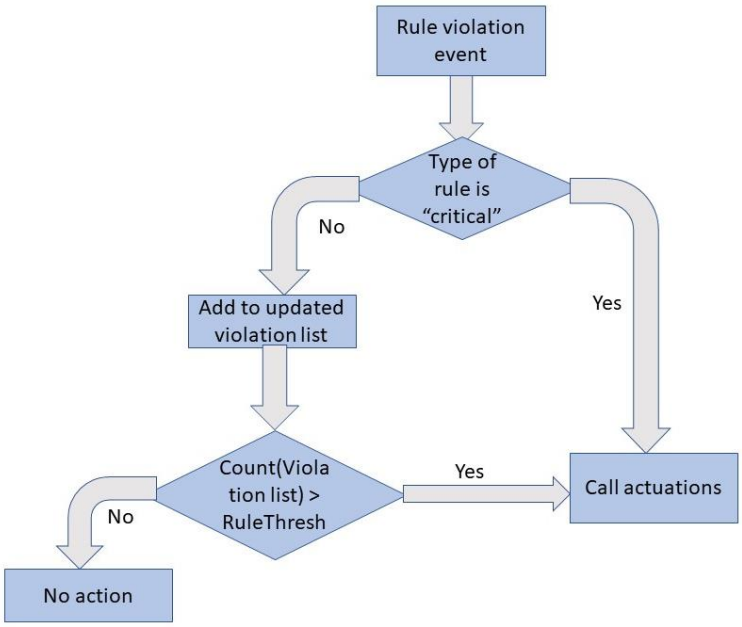

Figure 6.4: Reporting rule violation for an SLA Manager instance

Figure 6.4 shows the workflow of a reported rule violation for an SLA Manager instance. In the case of a critical violation, the SLA Manager instance calls the actuations immediately. However, in the case of a non-critical rule violation, the SLA Manager instance adds the reported violation to the violation list. If the number of violations exceeds the RuleThreshold, the actuations are triggered. It is worth mentioning that all actuations in the policy are called. In cases where different actuations are expected based on different rule violations, these are translated into different policies.

7. Conclusion

This report (i.e., the final report on the status of WP3 - part 2) concludes the efforts made by BRAINE WP3 partners in five areas: the design of a novel Cognitive Framework, the architecture of the Forecasting Functional Block (FFB), the challenges and a solution for deploying K8s in edge computing environments, the extension of the vocabulary to support workflow placement and description, and the design and implementation of the SLA Broker.

The information presented in this report is expected to be useful for researchers and practitioners in the field of edge computing, AI, and machine learning.