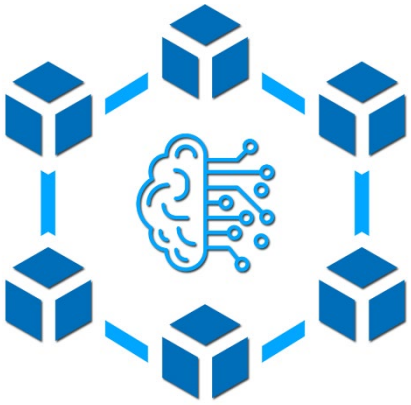

NANCY

An Artificial Intelligent Aided Unified Network for Secure Beyond 5G Long Term Evolution [GA: 101096456]

Deliverable 3.3

NANCY AI-based B-RAN Orchestration

Programme: HORIZON-JU-SNS-2022-STREAM-A-01-06

Start Date: 01 January 2023

Duration: 36 Months

Co-funded by the European Union

6GSNS

NANCY project has received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101096456.

Document Control Page

Deliverable Name	NANCY AI-based B-RAN Orchestration
Deliverable Number	D3.3
Work Package	WP3
Associated Task	T3.3 AI-Based B-RAN Orchestration Functionalities & Self-Evolving AI Model Repository Functionalities
Dissemination Level	Public
Due Date	M24
Completion Date	23 December 2024
Submission Date	24 December 2024
Deliverable Lead Partner	CERTH
Deliverable Author(s)	George-Nektarios Panayotidis (CERTH), Theofanis Xifilidis (CERTH), Katerina Politikou (CERTH), Dimitrios Kavallieros (CERTH), Andrej Cop (JSI), Blaz Bertalanic (JSI), Carolina Fortuna (JSI), Shih-Kai Chou (JSI), Thanasis Tziouvaras (Bi2S), Hatim Chergui (I2CAT), Miguel Catalan (I2CAT),Daniel Casini (SSS), Luca Abeni (SSS), Mauro Marinoni (SSS), Alessandro Biondi (SSS), Panos Matzakos (INTRA), Konstantinos Kyranou (SID), Georgios Niotis (SID), Georgios Michoulis (SID), Georgios Tziolas (SID)
Version	1.0

Document History

Version	Date	Change History	Author(s)	Organisation
0.1	08/09/2024	Section 3 content and subsection 5.1 content	Andrej Cop, Blaz Bertalanic, Carolina Fortuna	JSI
0.2	12/09/2024	Added content to Section 1 (Introduction)	George-Nektarios Panayotidis, Theofanis Xifilidis, Katerina Politikou	CERTH
0.3	18/09/2024	Added content to subsections 3.1.6, 3.3.4.	Thanasis Tziouvaras	Bi2S
0.4	28/09/2024	Section 2 content	Hatim Chergui	I2CAT
0.5	01/10/2024	Section 6 content	Daniel Casini, Luca Abeni, Mauro Marinoni, Alessandro Biondi	SSS

0.6	09/10/2024	Section 7 content	Panos Matzakos	INTRA
0.7	20/10/2024	Added content to subsection 5.2	Hatim Chergui, Miguel Catalan	I2CAT
0.8	30/10/2024	Added content to Executive Summary and Conclusions	George-Nektarios Panayotidis, Theofanis Xifilidis, Katerina Politikou	CERTH
0.85	30/10/2024	Section 4 content	Shih-Kai Chou	JSI
0.9	30/10/2024	Fixed figure captions and figure references, completed acronym list, reference list and in-text references	George-Nektarios Panayotidis, Theofanis Xifilidis, Katerina Politikou	CERTH
0.95	22/11/2024	Completed internal review by SIDROCO	Konstantinos Kyranou, Georgios Niotis, Georgios Michoulis, Georgios Tziolas	SID
0.97	30/11/2024	Completed internal review by CERTH	George-Nektarios Panayotidis, Theofanis Xifilidis, Dimitrios Kavallieros	CERTH
0.98	13/12/2024	Completed addressing quality check remarks	Panos Matzakos, Hatim Chergui, Blaz Bertalanic, Shih-Kai Chou, Daniel Casini	I2CAT, INTRA, SSS, JSI
1.0	18/12/2024	Quality Revision	George-Nektarios Panayotidis, Theofanis Xifilidis, Dimitrios Kavallieros	CERTH

Internal Review History

Name	Organisation	Date
Kyranou Konstantinos	SIDROCO Holding Ltd.	20 November 2024
Giorgos-Nektarios Panayotidis, Theofanis Xifilidis, Dimitrios Kavallieros	CERTH	30 November 2024

Quality Manager Revision

Name	Organisation	Date
Anna Triantafyllou, Dimitrios Pliatsios	UOWM	23 December 2024

Table of Contents

List of Figures

List of Acronyms

Acronym	Explanation
5GPPP	5G Public Private Partnership
AI	Artificial Intelligence
API	Application Programming Interface
ARFCN	Absolute Radio Frequency Channel Number
B5G	Beyond 5G
BLE	Bluetooth Low Energy
B-RAN	Blockchain Radio Access Network
CBS	Constant Bandwidth Server
CDI	Container Device Interface
CI/CD	Continuous Integration/Continuous Deployment
CNF	Containerized Network Function
CPU	Central Processing Unit
CSF	Compositional Scheduling Framework
CU	Central Unit
DAG	Direct Acyclic Graph
DoF	Degrees of Freedom
DRA	Dynamic Resource Allocation
DU	Distributed Unit
EDF	Earliest Deadline First
ETSI	European Telecommunications Standards Institute
FLOPs	Floating Point Operations Per Cycle Per Core
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
IP	Internet Protocol
K8s	Kubernetes
KL-Divergence	Kullback–Leibler Divergence
MEC	Mobile Edge Computing
ML	Machine Learning
ML Ops	Machine Learning Operations
MLP	Multi-Layer Perceptron
NF(s)	Network Function(s)
NS(s)	Network Service(s)
O-RAN	Open Radio Access Network
OSM	Open Source Management orchestration
PRB	Physical Resource Block
QoE	Quality of Experience
QoS	Quality of Service
RBAC	Role-Based Access Control
REST	REpresentational State Transfer
RSS	Received Signal Strength
RU	Radio Unit
STD	Standard Deviation Difference
SEMR	Self-Evolving Model Repository
SM	Slice Manager
TPU	Tensor Processing Unit
UE	User Equipment
VLAN	Virtual Local Area Network

XGB	eXtreme Gradient Boosting

Executive summary

This document focuses on the identification of the AI functionalities that will be incorporated in the NANCY architecture, so as to support the B-RAN orchestration framework and serve as an instrumental force in exploiting the massive set of settings and possibilities or, in more technical terms, its infinite Degrees of Freedom (DoF). The overall architecture takes advantage of the loose coupling encountered in service-based structures.

Specifically, in section 2, the deliverable describes the AI-based B-RAN orchestration functionalities that mainly concern the creation and management of network slices, as combinations of compute chunks. The position of the slice manager within the NANCY architecture is clearly specified and detailed instructions are given as to how to deploy the manager on a Kubernetes cluster.

In section 3, the Self-Evolving Model Repository (SEMR) functionalities are presented, namely data storage and preparation, model training, model storage, model deployment and inference, and the monitoring of the repository during the operation, with triggers for automated model retraining. In addition, the integration and implementation details of SEMR are provided via suitable code snippets. The purpose of the aforementioned repository is to leverage fresh data to trigger model retraining procedures with the aim of responding to changing network conditions as needed and improving the overall performance. Two methods are proposed for triggering the model retraining operation: a proactive and a reactive one. This section is completed with the comparison of the proposed SEMR solution with the O-RAN solution under different sizes of manipulated datasets.

In section 4, our innovative, recently introduced metric, named eCAL, is used, in order to quantify the energy efficiency of an AI/ML model. eCAL provides a thorough insight into sustainability and it is defined as the ratio of the total energy consumed by data manipulation components as per the totality of bits manipulated during the entire AI lifecycle, from data collection to model inference. The impact of each process of the AI lifecycle on the energy consumption is examined and the energy efficiency of a random AI/ML model is evaluated.

Section 5 elaborates on the interconnection of AI and SEMR. The requirements for instantiating SEMR on a network are described. An interconnection scenario of SEMR and SM in the event of detecting model retraining necessity is analyzed providing a clear picture of SEMR operational cycle relative to model retraining.

Then, section 6 presents a framework we developed to schedule Kubernetes containers by leveraging the SCHED_DEADLINE scheduler to achieve adequate QoS and, especially, timing isolation. Furthermore, thanks to the usage of the KubeRay implementation environment, Kubernetes is also compatible with Ray, a state-of-the-art framework for parallel workloads that supports distributed machine learning. The mainline SCHED_DEADLINE Linux scheduler has been extended to be compatible with containers. Similarly, a Kubernetes plugin has been developed to allow the containerization stack to interact with SCHED_DEADLINE, e.g., by modifying the “kube-scheduler” and “kubelet” elements. Hence, the new, complete RT-Kubernetes architecture is overall new and novel and one that hinges on the so-called Dynamic Resource Allocation (DRA) –driver for interacting with Kubernetes component and its role as admission control preserving only worker nodes suitable for hosting real-time containers. This last element allows for making accurate claims regarding real-time containers, such as the number of CPU cores, the runtime, and the reservation period; these requirements may be thoroughly depicted in a YAML file.

Finally, section 7 concerns the integration of the above-mentioned functionalities into the NANCY platform. At first, a general overview of the NANCY CI/CD (Continuous Integration/Continuous

Delivery) infrastructure and its open-source DevOps services is provided. The rest of this section focuses on the deployment and testing of the SM and SEMR components in a common environment, rendering their interconnection seamless. Specifically, for both components, dedicated Jenkins and Harbor workspaces are created in order to create and manage their CI/CD workflow and handle their container images, respectively.

1. Introduction

WP3 provides the overall NANCY architecture which is based on three pillars: (a) the incorporation of O-RAN and prior 5GPPP project architectural design findings (network slicing, edge computing, service-based structure etc.); (b) the development of NANCY-enabling innovations and their integration into the architecture; and (c) Mobile Edge Computing (MEC). Overall, this work package pursues six objectives as follows:

a) identify the architectural gaps in current research projects and state-of-the-art (SOTA) solutions;
b) identify the key outcomes and the architecture commonalities from O-RAN solution;
c) specify the required architectural components to support B-RAN;
d) define in detail the overall NANCY reference architecture, including the software framework, tools, schemes, and algorithms by taking into account the identified requirement in WP1, along with the current SOTA technology axes, models and O-RAN open architecture requirements;
e) design novel (AI-based) algorithms, functionalities and solutions following the experimental-driven modelling and optimization approach;
f) identify and specify orchestration functions that will be used to manage the overall orchestration framework and support the dynamic nature of NANCY.

Deliverable 3.3 mainly focuses on objective (f) "*identify and specify orchestration functions that will be used to manage the overall orchestration framework and support the dynamic nature of NANCY*" and reports the results of the corresponding Task 3.3 which concerns two main NANCY project results, namely [R7] and [R9]. Specifically, result [R7] is related to the augmentation of the slice manager, that exploits the blockchain technology, while also supporting AI functionalities which contribute to the creation and configuration of network slices and the continuous improvement of various B-RAN characteristics such as throughput, overhead times and more. Finally, regarding the result [R9], this refers to the design of a novel self-evolving AI model repository. In this deliverable, an overview of the aforementioned innovations will be presented, along with their specific positioning within the NANCY architecture and followed by their implementation and evaluation.

Receiving the specific use cases and overall NANCY architecture design as inputs, the work of D3.3 mainly leverages AI to provide orchestration in the B-RAN infinite Degrees of Freedom (DoF) context. The orchestrator can effectively coordinate the interaction of various network users and services, manage data flow and optimize the use of computational resources in a secure environment. Going beyond the identified use cases and their explicit characteristics, a generalized methodology will be applied to support emerging services based on initialization and instantiation of network slices in a flexible, elastic and secure manner. To that end, the decomposition and allocation of Network Functions (NFs), allows resulting function groups to be autonomously allocated in the cloud, edge or user plane [1]. Depending on the computational resource availability, mobility patterns and multiple roles each mobile node can play, NFs allocation will be determined by the requirements identified for the MEC and Slices [2]. In this context, performance indicators that must be met such as stringent latency require that NFs must be located at the individual infrastructure at the edge as opposed to computation-intensive tasks where NFs must be allocated to the edge or even in the cloud plane. The complexity of the B-RAN network in terms of dynamic topology, node mobility and diverse service requirements, together with time-varying resource availability and requirements, dictate the applicability of AI, in order to accurately perform user association, routing and resource allocation and optimize decentralized B-RAN performance. The problem of massive data exchange requirements and

slow network convergence is addressed in NANCY by employing transfer learning to migrate knowledge of task execution in related tasks, to reduce network convergence time.

Moreover, a self-evolving model repository (SEMR) is of much importance for keeping up with the new data that become available and retaining an up-to-the-minute AI software component. This repository is closely interwoven with the Continuous Integration/Continuous Deployment (CI/CD) process, also in place in NANCY. This repository is to keep up with developments and invariably better oneself via continuous training, which implies already having a final piece of software and at the same time building upon it, retraining and thus keeping its edges sharp. The human intervention is meant to be significantly lessened. By incorporating certain algorithmic aspects, NANCY introduces a novel approach that enables the repository to become independent of human design, allowing it to adapt to dynamic conditions and requirements. The repository is meant to contain component functions for storing and searching models based on optimization procedures and initializing parameters leveraging low-complexity mathematical operations as its building blocks [3].

In the evolving 5G intelligent radio access networks, energy efficiency continues to emerge as a vital metric due to massive connectivity and decentralized network architecture along with diverse service requirements. Moreover, the extended use of AI/ML processes, along with its various benefits, also involves the risk of degraded energy efficiency. In particular, during the training phase of an AI model, significant computational power is demanded for the collection, storage and manipulation of extensive datasets, while, at the same time, the inference stage is an ongoing operation that consumes a non-negligible amount of energy. It is thus obvious that any AI task is prone to be energy-intensive. When integrating AI/ML sub-systems in the NANCY architecture, it is necessary to account for energy efficiency. In this way, we can design sustainable models with lower energy consumption, while maintaining a satisfying operating efficiency.

Regarding the interconnection of SEMR and AI, the latter will boost automation and reduce human effort relative to repository functionalities allowing low-complexity and secure transfer of AI model between different verticals. Thus, model (re)training, storage as well as deployment as a service will be rendered viable. With this interoperable AI approach, models can be re-instantiated, accounting for dynamic topology and computational resource requirements. Leveraging AI/ML workflow in large datasets, the execution time of the proposed SEMR in tasks such as data extraction and model training and deployment can be significantly reduced, which is proven more pronounced in scenarios where the self-evolving property of model repository will be aligned with slice instantiation, network functions isolation and efficient resource utilization.

Workload scheduling is a feature that has been already in place in cellular networking even before the advent of 5G [4]. Schedulers are meant to provide a fair and efficient distribution of the computational load among network nodes, such as workstations and servers, in order to avoid resource starvation, reduce response time, increase reliability and thus ensure an overall improved user experience. In the realm of B5G and in the NANCY context, workload scheduling, combined with the key technology enablers of network slicing and mobile edge computing, renders partial offloading of computational load to edge devices a viable solution, in case an application has to function under stringent latency terms according to the Service-Level Agreements [5]. The answer to whether offloading is going to be implemented or not, as well as which part of the computational load it would concern and to which devices it is to be offloaded, is provided by the solution of an optimization problem which involves dynamic features such as task computation resource requirements, minimum execution time, task storage in edge devices and device residence time in coverage areas. In NANCY B-RAN architecture, real-time AI-driven decision-making techniques are exploited, to further improve the solution

optimality of the aforementioned optimization problem and ensure that the workload scheduling is completed in an efficient, flexible and secure manner.

The last part of this deliverable describes the integration of these previously mentioned functionalities in the NANCY platform. At a high level, the NANCY platform is composed of three entities – the CI/CD toolbox, the development/testing environment and the production/demonstration environment. The CI/CD tools, namely GitHub, Jenkins etc., exploit the best practices for software containerization and support the entire software lifecycle processes up to the release of a fully tested and deployed operation system. Specifically, during the integration process, the developer pushes his/her code into the project's GitHub repository and then Jenkins triggers the automated built of the component and the creation of an image which can later be stored in a container repository. Then, unit, functional and integration tests designed for the specific component are executed in a Kubernetes environment to ensure fine operation before proceeding with the demonstration environment. Given that all the individual tests are successful, the developer's code merges with a master branch that is stored in the repository. Then, similarly to the testing process, CI is triggered, the image is pulled from the registry and the new component is deployed in the Kubernetes production environment.

2. AI-based B-RAN Orchestration Functionalities

2.1. Overview of the Functionalities

The slice composition workflow in ETSI provides a structured approach for creating and managing network slices, crucial for dynamic and efficient network management. A specific tool that implements this process is the Slice Manager (SM), which follows a detailed sequence of steps (Figures 1 to 4). The workflow begins with (i) creating compute resources and chunks, which involves adding a Kubernetes (K8s) cluster to the SM via the corresponding `Kubeconfig` file, checking connectivity and credentials as well as fetching information about the resources and storing it in its database. A compute chunk in this case is a K8s namespace with specific resources (computing, memory and storage). Next, (ii) network resources and chunks are established, ensuring that the slice has the required connectivity infrastructure in the form of a VLAN. Following this, (iii) radio resources and chunks are created and, the necessary wireless capabilities are defined, including physical resource blocks (PRBs) and cell frequency (ARFCN). After all resources are prepared, the process moves to (iv) slice creation, which logically integrates these chunks into a cohesive network slice via their `chunk_ids`. Once the slice is created, it is (v) activated and becomes, operational and ready for use. Finally, (vi) the application is instantiated within the slice after onboarding it on the Open Source Management (OSM) orchestration, enabling specific services and functionalities tailored to the network's requirements. This structured approach ensures that each aspect of the network slice is carefully planned and executed, providing robust and scalable network management solutions.

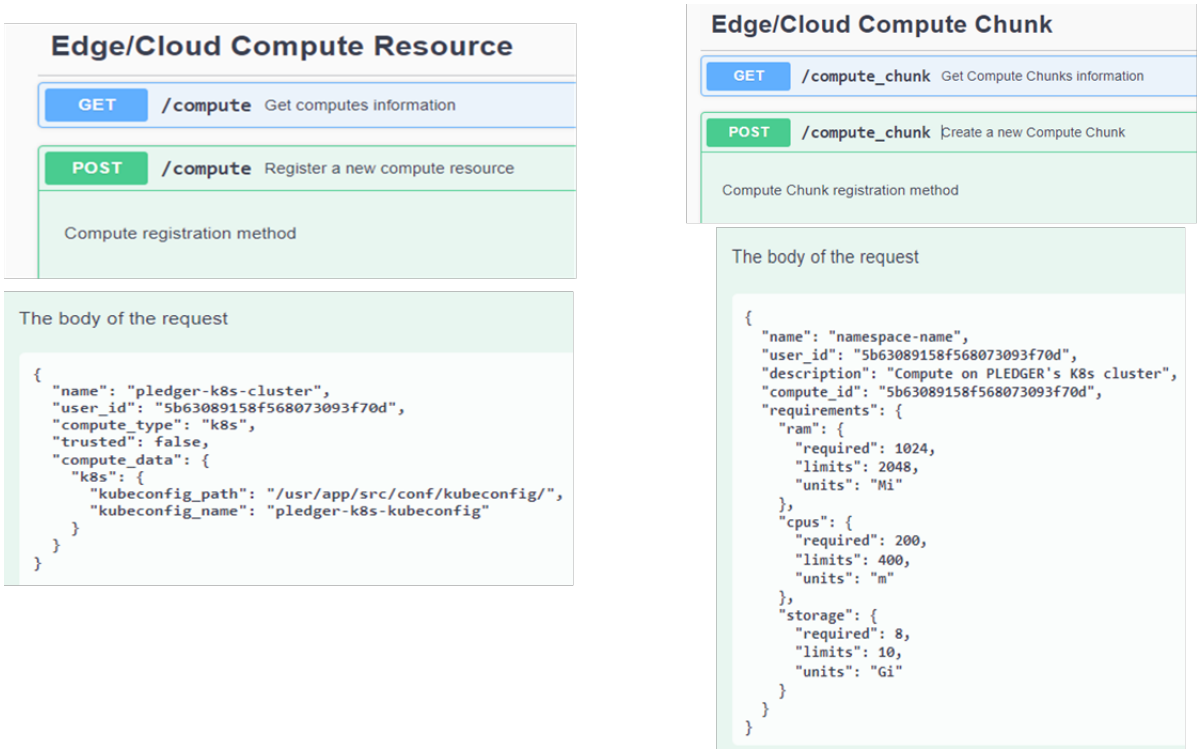

Figure 1. Adding a compute resource and chunk to the Slice Manager

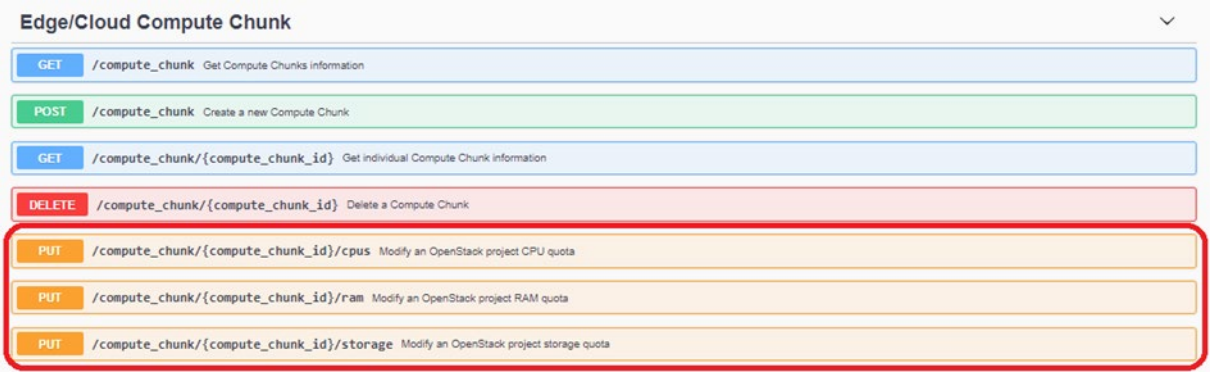

Figure 2. Edge/Cloud Chunk API

Note that a slice can be created with N compute chunks that can be instantiated in different clusters. That way, by having a slice containing compute chunks from different clusters, we could instantiate services working in a multi-cluster environment.

Created Slices can be reconfigured by adding/removing chunks (referred by `chunk_id`), following a set of requirements for slice composition/update logic (for instance, every instance must have at least one compute chunk; network chunk of type data-network is compulsory for slices with K8s compute chunk; network chunk of type access-network is compulsory when the slice includes cellular access; only one radio chunk is admissible per slice; chunks in use cannot be deleted).

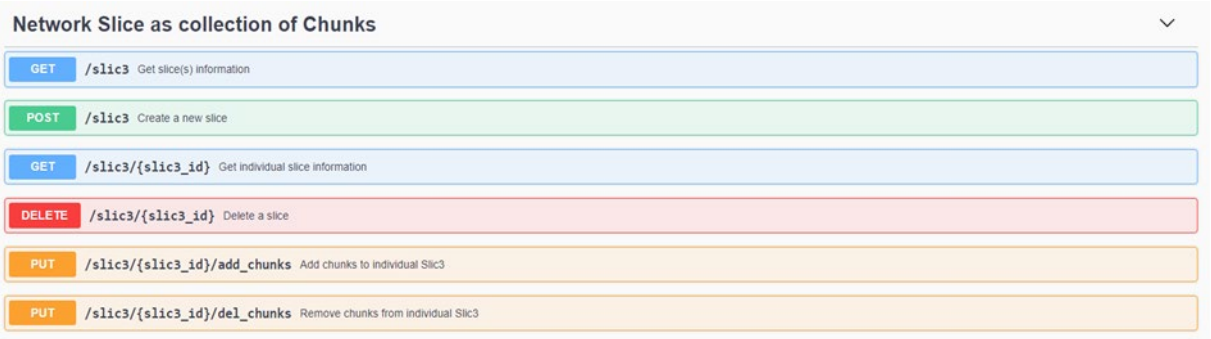

Figure 3. Network Slice API

Next, the App to be orchestrated is instantiated on top of the created slice. It is assumed that it is already onboarded on OSM.

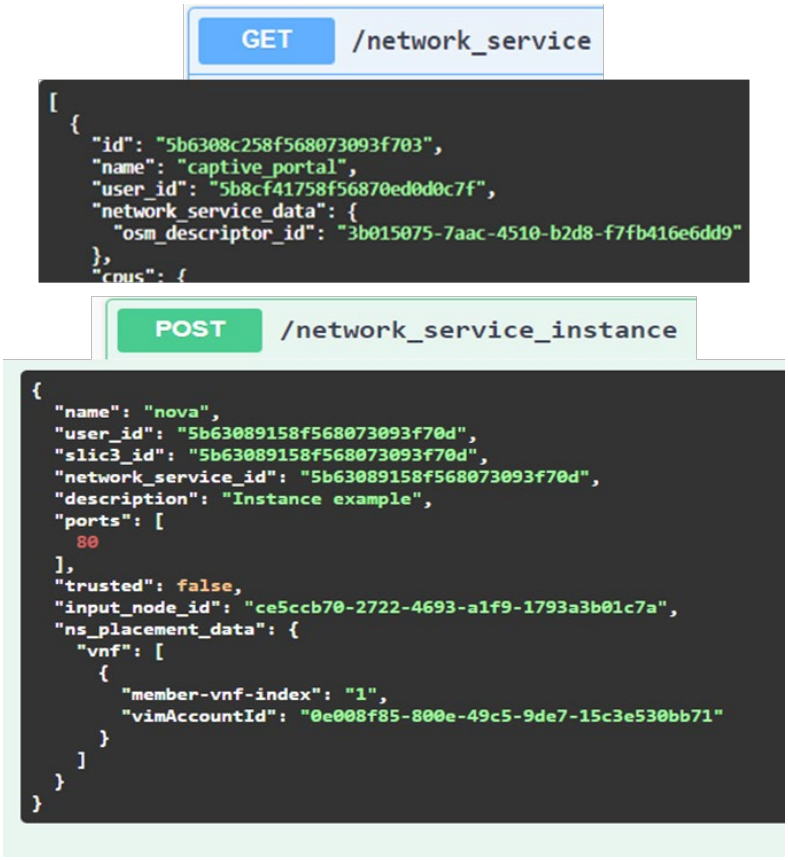

Figure 4. App instantiation on top of slice

2.2. Positioning within the NANCY architecture

As an orchestrator, the Slice Manager is positioned within the NANCY architecture in the enforcement block, as every AI-based decision engine needs it to actuate on the infrastructure, reconfiguring e.g., resources allocated to a specific slice as shown in Figure 5.

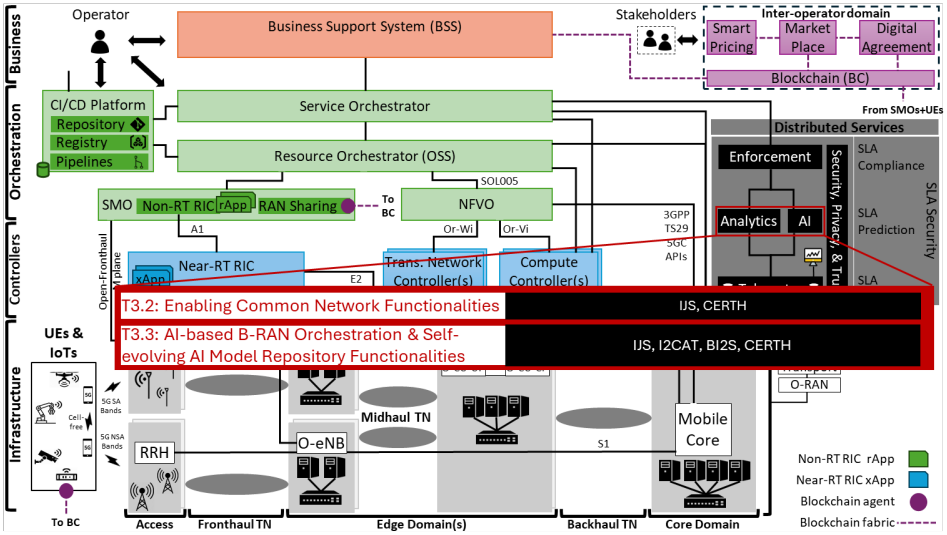

Figure 5 Positioning within the NANCY architecture

2.3. Implementation

To deploy the Slice Manager on a K8s cluster, the following steps should be followed:

- Using the following commands, create a namespace and docker-registry secret for pulling the image (Another option is to build the docker image locally. To build the docker image and create a container Docker is required)

```
$ kubectl create namespace soe
$ kubectl create -n soe secret docker-registry sm-regcred\
      --docker-server=<REGISTRY_URL>\
      --docker-username=<REGISTRY_USERNAME>\
      --docker-password=<REGISTRY_PASSWORD>
```

- Next, generate the `config.cfg`, `quota.cfg` and `kubeconfig` files relative to the target K8s environment and create a secret out of them, using the command below

```
$kubectl create -n soe secret generic sm-config-quota-
kubeconfig\
      --from-file=</path/to/config.cfg>\
      --from-file=</path/to/quota.cfg>\
      --from-file=</path/to/kubeconfig>
```

In a Kubernetes (K8s) environment, configuration files like `config.cfg`, `quota.cfg`, and `kubeconfig` play crucial roles in defining the operational parameters, resource limitations, and cluster access settings, respectively. The `config.cfg` file typically contains various configuration parameters essential for the deployment and management of services within the cluster, such as environment variables, resource specifications, and service endpoints. The `quota.cfg` file is used to define resource quotas and limitations, ensuring fair and controlled allocation of CPU, memory, and storage across different namespaces, thus preventing any single workload from monopolizing resources. The `kubeconfig` file, on the other hand, is vital for managing cluster access. It stores the necessary credentials and context information for connecting to one or more Kubernetes clusters, enabling users and tools to authenticate and interact with the cluster securely. Together, these files provide a comprehensive framework

for efficiently managing configurations, allocating resources, and controlling in a Kubernetes ecosystem, ensuring optimal performance and security.

- Finally, edit the file deploy/sm-svc-deploy.yaml by updating the image name in the container registry (field image in container specs) and apply this file with the following command

```
$ kubectl apply –f deploy/sm-svc-deploy.yaml
```

- Once the server is running, please check the API Swagger documentation by visiting the link: http://localhost:8989/(server's default configuration sample). The API Swagger documentation on port 8989 serves as an interactive interface for developers to explore and interact with the API endpoints provided by a service. Swagger, now part of the OpenAPI specification, offers a standardized way to describe the structure of an API, including its endpoints, request and response formats, parameters, and authentication methods. By hosting Swagger documentation on port 8989, the service provides a user-friendly web interface where developers can view detailed information about available API calls, test them directly within the browser, and see real-time responses. This documentation is invaluable for both developers and integrators, as it enhances understanding, facilitates testing, and ensures consistency in how the API is used and integrated into applications.

3. Self-Evolving Model Repository Functionalities

3.1. Overview of Functionality

The SEMR provides essential functionalities for machine learning automation, known as Machine Learning Operations (MLOps) [6], which aim to automate and enhance the development, training, and storage of machine learning models. Self-evolving, in this context, refers to the ability of AI models to improve their performance over time by incorporating new data. This capability is particularly important for dynamic processes like the management and orchestration functionalities of the control plane, which must continually adjust to network load. By enabling models to retrain on fresh data, SEMR ensures that AI systems can adapt to changing network conditions and maintain optimal performance.

Additionally, deploying SEMR on the network edge offers several advantages. It allows specialized models and services to be positioned closer to users, optimizing their latency while enabling these deployments to scale and offload onto central units as needed. The ability to fluidly retrain and update models with diverse datasets creates truly self-evolving solutions [7] for controlling AI-native access networks. This approach enhances the adaptability and versatility of existing AI/ML systems, ensuring that the most appropriate model is always utilized for a given task or scenario as the network environment evolves.

While current software for automating AI/ML workflows in Open Radio Access Network (O-RAN) [8] addresses some challenges, it lacks maturity and ease of use. SEMR adheres to the architectural principles and specifications defined by the O-RAN Alliance [9] while incorporating key features typically found in democratized and open software. These include support for various processor architectures, customization options, scalability, and distributed services [10].

Specifically, the NANCY Self-Evolving AI Model Repository is responsible for storing and searching for AI models, managing the deployment of AI/ML models as a service, and retraining models on updated datasets, thus enabling a self-evolving repository for network services.

SEMR supports five main functionalities: data storage and preparation, model training, model storage, model deployment, inference and continuous operation, and proactive retraining of the models.

3.1.1. Data Storage and Preparation

The SEMR data storage and preparation functionalities ensure that all necessary data is cleaned, transformed, organized, and ready for training AI/ML models. Data transformation procedures are integrated into SEMR, collecting data from the network, and storing the transformed data in SEMR's data storage.

3.1.2. Model Training

The model training functionalities enable both the original training and continuous retraining of AI/ML models across heterogeneous network infrastructure. This utilizes edge and central units to distribute the computational load of model training with new data from the SEMR data storage solution, ensuring the highest-performing models and services are available whenever needed. Existing model training procedures can be configured to align with SEMR’s templates for model training and retraining. The code must be adapted to SEMR’s training tools and configured to utilize the required computational resources. Once trained, models are autonomously registered and versioned in the model repository for deployment and future retraining.

3.1.3. Model Storage

Model storage is the core functionality of SEMR, providing access to all versions of models. Models can be accessed by name and version, either by SEMR or other users of the system. SEMR offers storage for all AI/ML models for network use cases and can act as a model storage solution or as part of a full SEMR pipeline for training and retraining models.

3.1.4. Model Deployment and Inference

SEMR enables the deployment of AI/ML models as a service. Using SEMR's model inference charts, models can be instantiated and configured, specifying the model version, Docker image, service port, number of replicas, and other configurations required by the service. When a new model version is uploaded to SEMR (e.g., when models are retrained), the inference service can be re-instantiated with new configurations.

3.1.5. Continuous Operation and Monitoring

Enabled by continuous operation and monitoring functionalities, SEMR offers proactive and reactive analysis of input data and model drift to continuously improve and retrain models. SEMR follows the principles of Direct Acyclic Graphs (DAGs) for defining AI/ML tasks and their connections and can schedule tasks like model performance monitoring to trigger appropriate events, such as model retraining and redeployment.

3.1.6. Self-evolution through reactive and proactive retrain trigger

SEMR offers a service for automating the AI/ML model retraining process. The retraining operation is initiated when a “significant” deviation is observed within the training dataset. To support this feature, NANCY deploys reactive and proactive triggering mechanisms which monitor the AI/ML training data and fire a “retraining” signal when necessary. This approach guarantees that the model quality and accuracy remain at high levels and do not degrade over time.

3.2. Positioning within the NANCY Architecture

As an AI model orchestrator, the SEMR is positioned within the NANCY architecture in the Enforcement block, similar to SM and Analytics as seen in Figure 5. The SEMR takes care of the lifecycle of the models, monitoring them and enforcing their retraining based on the proactive and reactive policies within the SEMR.

3.3. Implementation

SEMR is implemented using state-of-the-art open-source tools for its components. Components are orchestrated, virtualized and managed utilizing Kubernetes container orchestration software. The implementation is open-sourced and is available in the public GitHub repository.

3.3.1. SEMR Deployment

SEMR is capable of deployment on heterogeneous Kubernetes clusters that include both ARM and x86_64 GNU/Linux nodes. This configuration simulates deployment across distributed edge infrastructure, network slices in Radio Access Networks [11], and Multi-access Edge Computing (MEC) devices, which provide computing functionalities near end users at the network edge [12]. Additionally, SEMR can utilize the computational resources of the entire distributed cluster for various stages of the AI/ML workflow such as distributed model training and distributed model inference.

Deployment is supported through helm charts and can be configured using helm values. SEMR components can be enabled, disabled and be allocated the appropriate amount of cluster resources, depending on the use case and deployment infrastructure. Example helm values for SEMR deployment are provided in the next code snippet (Figure 6):

```yaml
general:
  ip: "<SEMR_IP>"

# Ray
workerGroups:
  amdGroup:
    replicas: 1
    container:
      resources:
        limits:
          cpu: "2"
          memory: "4G"
        requests:
          cpu: "2"
          memory: "4G"

# MLflow
mlflow:
  enabled: true

# MinIO
minio:
  enabled: true

# Flyte
flyte-binary:
  enabled: true
  configuration:
    storage:
      providerConfig:
        s3:
          endpoint: "http://<SEMR_IP>:30085"

# promethes&grafana
kube-prometheus-stack:
  enabled: true
```

Figure 6. Code Snippet with example helm values for SEMR deployment

3.3.2. SEMR Architecture

The architecture of SEMR follows an established AI/ML workflow architecture for Open RAN [4] and is presented in Figure 7.

The SEMR architecture presents open-source tools selected for different AI/ML components and the relationships between them. Figure 7 shows a flow of operations and data artifacts within the SEMR architecture.

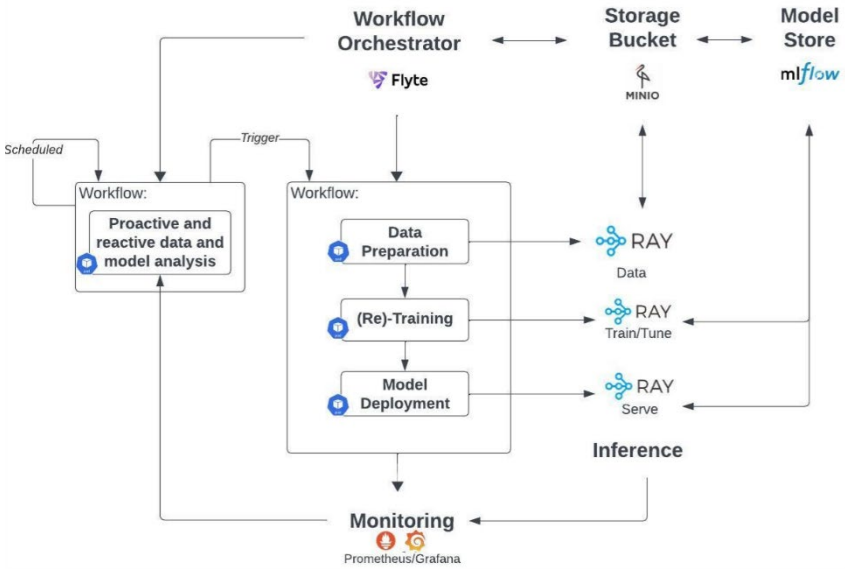

Figure 7. SEMR Architecture Diagram

For SEMR data storage, the MinIO storage bucket is utilized. It is accessible by API on http://<SEMR_IP>:30090/ IP address and data can be inserted or read with the following python code to perform data collection and preparation (Figure 8):

```
import s3fs
import pandas as pd

s3_fs = s3fs.S3FileSystem(
    key='minio',
    secret='<password>',
    endpoint_url=f'http://<SEMR_IP>:30085',
)

# Specify the path to the CSV file in MinIO
file_path = f'raybuck/qoe_data/liveCell-x10.csv'

# Use pandas to read the CSV file directly from MinIO
with s3_fs.open(file_path, 'rb') as f:
    df = pd.read_csv(f)
```

Figure 8. Python code for data collection and preparation

Ray framework and its ML libraries (Data, Train, Tune, Serve) are utilized for model training. Training jobs or other python tasks can be submitted to the Ray cluster using an API that is accessible on http://<SEMR_IP>/ray/ IP address. SEMR supports distributed model training through the Ray

framework. It is framework-agnostic and, with little to no modification, supports the execution of existing machine learning training procedures with frameworks like Keras, TensorFlow, and PyTorch.

MLflow is used as a model store to support efficient model storage that can provide actors outside the AI/ML workflow cluster with access to the latest models. The training component in the SEMR architecture can fetch models stored in MLflow for model retraining and store the updated models in MLflow, as depicted in the SEMR architecture in Figure 7. The MLflow model store is available at http://<SEMR_IP>:31007/#/models IP address. Models can be accessed through the API as presented in the code snippet of Figure 9:

```python
import mlflow

# SEMR's model store endpoint
os.environ['MLFLOW_TRACKING_URI'] = 'http://<SEMR_IP>:31007'

# Log trained ML model to SEMR
mlflow.pytorch.log_model(model, "CNN_spectrum", registered_model_name="CNN_spectrum")
```

Figure 9. Code snippet for storing a trained model

Trained models can be deployed and exposed as an API inference endpoint using FastAPI and docker containers. Models can be pulled from the MLflow model store by name and version and are set up to run inference predictions on incoming API requests. The procedure to deploy AI/ML models as API endpoints is shown in the following code snippet (Figure 10):

```python
# SEMR's model store endpoint
os.environ['MLFLOW_TRACKING_URI'] = 'http://<SEMR_IP>:31007'

# Model version from the env variable
model_version = os.getenv('MODEL_VERSION', "latest")

# Modify the model_uri to point to the correct model name
model_uri = f"models:/CNN_spectrum/{model_version}"

# Modify the mlflow loading function
model = mlflow.pytorch.load_model(model_uri)
model.eval()
print("Model loaded!")

@app.post("/")
async def echo(data: List[List[List[float]]]):
    # Data transformations
    tensor_data = torch.tensor(data)
    tensor_data = tensor_data.unsqueeze(0)

    # Inference
    cnn_labels_array = cnn_predict(tensor_data, model)
    counts = Counter(cnn_labels_array)

    return {"prediction": str(counts)}
```

Figure 10. Code snippet for accessing a trained model from the model repository and its subsequent deployment

SEMR uses Flyte[10] as a workflow orchestrator. By creating Flyte workflows for training and metrics analysis, an intelligent decision for triggering model retraining can be implemented. The Flyte dashboard and API are available on IP address http://<SEMR_IP>:31082/ and is illustrated in Figure 11.

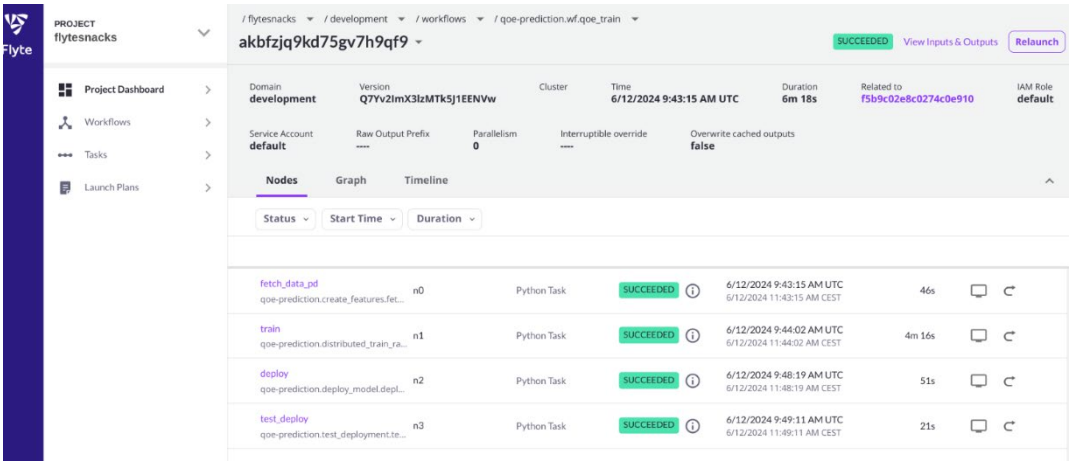

Figure 11. Flyte Workflow Orchestrator Dashboard

Prometheus[11] and Grafana[12] monitoring deployments are used to provide monitoring and metrics for intelligent retraining decisions. Grafana dashboard is exposed on the IP address http://<SEMR_IP>30000/ and Prometheus at IP address http://<SEMR_IP>:30002/. Prometheus metrics can be collected using the Prometheus API.

3.3.3. Utilization

The expected way for users to interact with SEMR is depicted in a user workflow diagram in Figure 12.

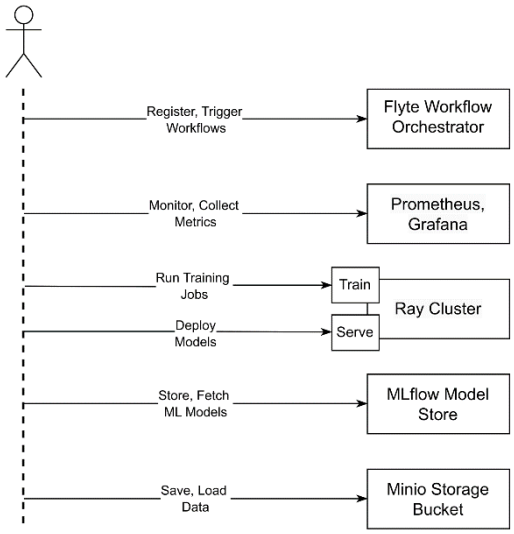

Figure 12. SEMR User Workflow Diagram

Each SEMR component is a separate module and can be utilized independently. Users can register AI/ML workflows with the SEMR Flyte orchestrator. AI/ML workflows are consecutive tasks to execute the entire ML procedure from data preparation to model deployment. Users can collect and monitor metrics through the Prometheus and Grafana deployments. Users can utilize the Ray computing cluster for running ML training jobs or serving ML models. The MLflow model store allows users to store and fetch ML models, either using the MLflow API or the exposed dashboard. Finally, the MinIO storage bucket endpoint is exposed to users for storing and retrieving the data for model training or any other use case.

3.3.4. Reactive and proactive trigger approach to model retraining

The SEMR supports both proactive and reactive methods for detecting data drift. Data drift is a phenomenon under which newly collected data diverge from older data and, thus, the AI/ML models trained with the old data underperform when using the newly collected data. As a result, a question arises as to "when to retrain an AI/ML model". Retraining the AI/ML model in frequent time intervals requires time, energy and computational resources and thus, it is considered suboptimal. Our goal is to find the optimal period in which the re-training operation ensures high model quality with the least possible iterations. For this reason, NANCY implements two approaches in SEMR, namely the "proactive approach" and the "reactive approach", with each approach being suitable for different data drift detection tasks.

Proactive approach: The proactive approach is illustrated in Figure 13. It does not require the trained ML/AI model to function; instead, it considers only the statistical information of the datasets. This makes the proactive approach a model-free method. Within this methodology, a set of statistical checks is conducted **(i)** between the old and new data; and **(ii)** between the extracted features of the old and new data. The feature extraction is realised through a Multilayer Perceptron (MLP), which is trained to extract high-level features from the input data. In the sequel, several metrics are calculated such as KL-divergence, mean value divergence, standard deviation difference, cross-correlation coefficient and Kolmogorov-Smirnov test, between **(i)** the datasets and **(ii)** the extracted features of data. The outputs of such statistical tests are forwarded to a utility function, which calculates the final utility of the model re-training operation. The higher the calculated utility, the more efficient it is to retrain the model.

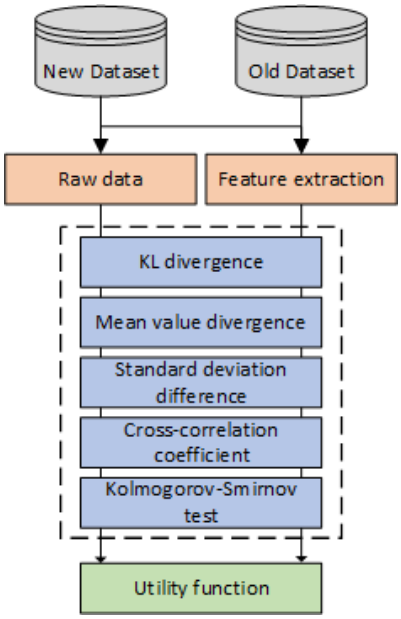

Figure 13. NANCY's SEMR proactive approach to data drift detection

Reactive approach: The reactive approach requires access to the trained AI/ML model and supports three distinct methods, as depicted in Figure 14. The user can select one of the provided methods or can opt for any combination of them. Below, we describe each of the three methods.

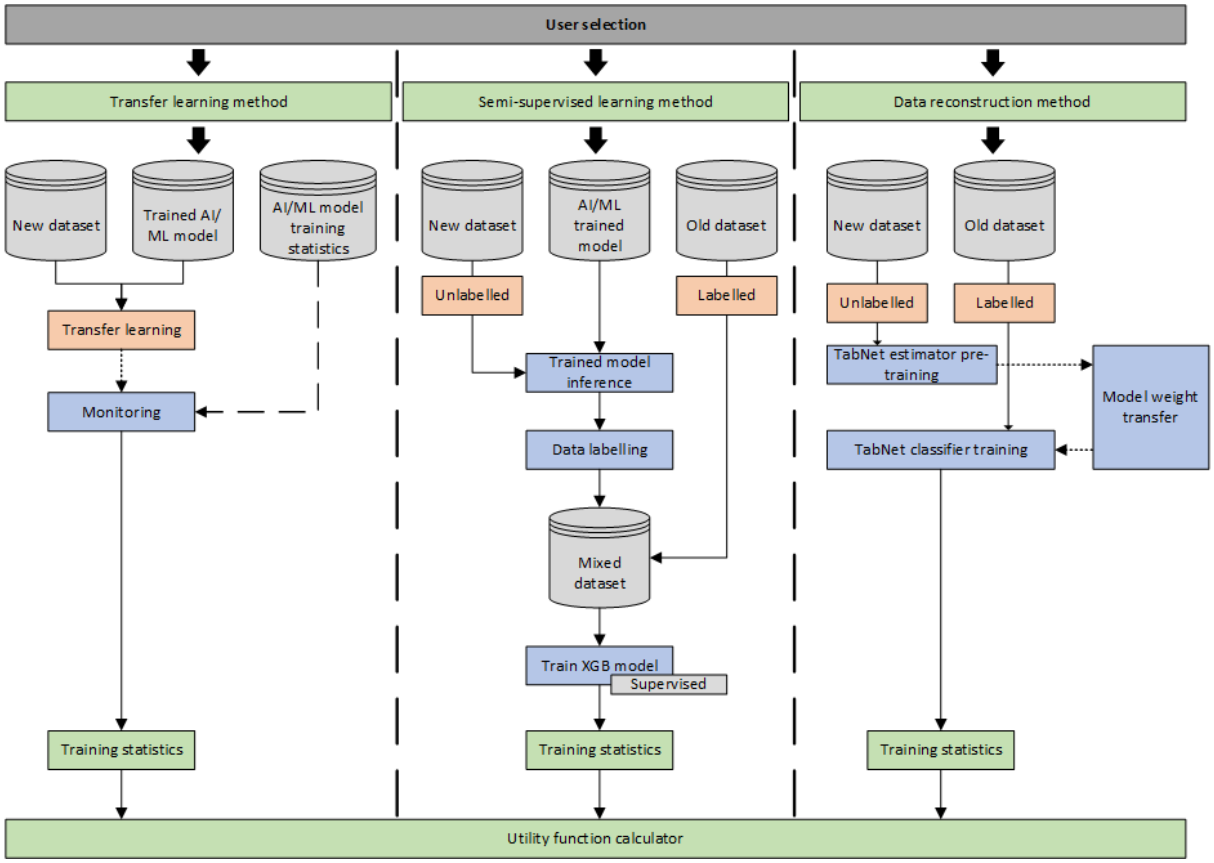

Figure 14. NANCY's SEMR reactive approach to data drift

- **Transfer learning method:** This method takes as input the newly collected data, the trained AI/ML model (which is trained with the old dataset) and the training statistics (loss and accuracy) of the corresponding model. A transfer learning operation is initiated, on the AI/ML model, using the new dataset. The transfer learning process runs for 3 epochs and then, the newly acquired training statistics are forwarded to the utility function calculator.
- **Semi-supervised learning method:** This method takes as input the old (labelled) dataset, the new (unlabelled) dataset and the AI/ML model, which is trained using the old data. It is suitable for cases when the newly acquired data are not labelled and thus, the data drift is hard to detect. The trained model is used to conduct an inference operation on the new dataset and data samples are labelled through this process. Then, a mixed dataset is formulated which contains the old data and the newly labelled data. This dataset is used to train an XGB classifier in a supervised fashion and the training statistics of the process are sent to the utility function calculator.
- **Data reconstruction method:** This approach leverages the new dataset, the old dataset and the TabNet model architecture [13]. Again, we consider that the new dataset contains unlabelled data, and the old dataset contains labelled data. A TabNet unsupervised estimator is pretrained using the unlabelled data (new dataset). In the sequel, the estimator's weights are transferred to a supervised TabNet model, suitable for performing classification tasks. This model is retrained using the old data and the statistics of the operation are broadcast to the utility function calculator.

The utility function calculator utilizes the available outputs of the methods described above to calculate the final utility. The calculator can function with any combination of the reactive methods and produces a utility score. The utility score is higher when the expected utility for re-training a model is high as well. Further, the training and pre-training operations conducted within the SEMR's reactive approach are designed to run for 2-3 epochs so that the requirements (in terms of time, energy and computational resources) are very small.

3.4. Evaluation

The deployment time of SEMR is compared to the O-RAN Software Community solution [14] in Figure 15. The testing infrastructure consisted of an Ubuntu Linux virtual machine with an Intel Xeon E5-2650 2.00 GHz CPU with 16 cores with one thread per core and 32 GB of RAM, and three edge devices with ARM-based Raspberry Pi 5 nodes with 4 CPU cores and 8 GB of RAM each. SEMR takes on average 939s (about 31 minutes) to deploy, while O-RAN solution takes 1577s (about 53 minutes). We can see that SEMR requires around 40% less time to deploy compared to O-RAN. As depicted in Figure 15, there is substantial variation in the deployment time for both solutions, with the standard deviation being around 270s and 100s for SEMR and O-RAN's solution, respectively.

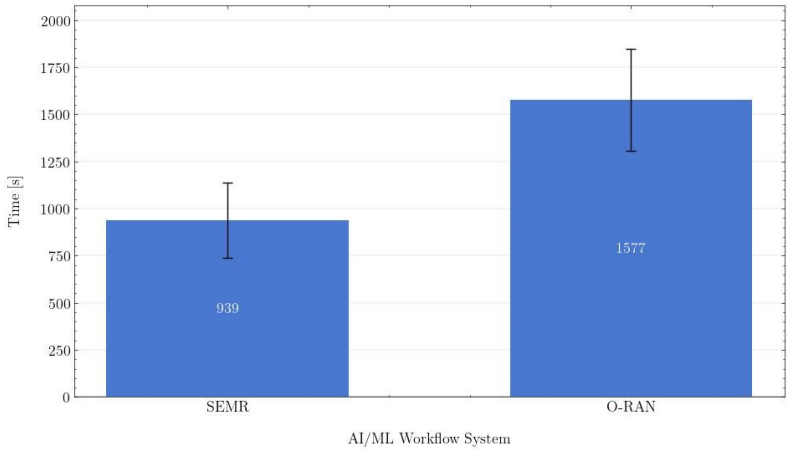

Figure 15. Deployment times of the SEMR and O-RAN solution

The AI/ML workflow execution time under the two approaches is presented in Figure 16. We tested O-RAN's example Quality of Experience (QoE) prediction workflow on three different dataset sizes and measured the execution time for each of the AI/ML workflow stages, comparing SEMR and O-RANsolution. For the smallest dataset size (1x) O-RAN performs around 24% (Δ67s) better than SEMR. For 10x dataset size the workflow execution time is around 73% (Δ965s) faster on SEMR and for 100x dataset size the workflow did not complete on the O-RAN solution.

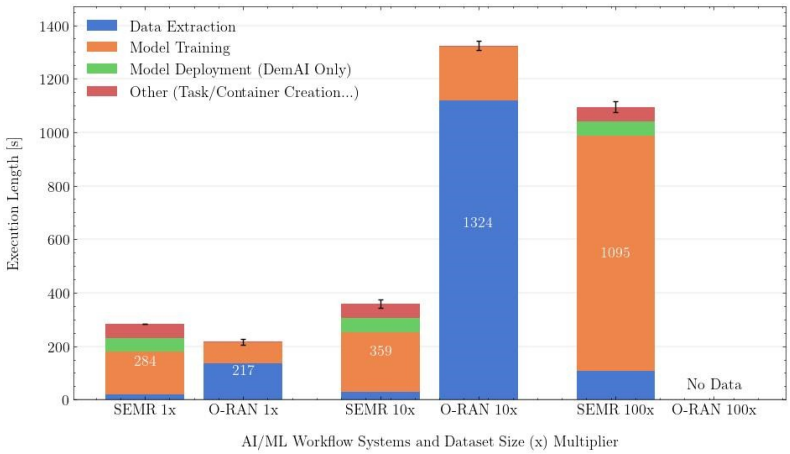

Figure 16: AI/ML Workflow Execution Time

The model training stage of the AI/ML workflow (colored in orange in Figure 16) is around 50% faster on O-RAN for 1x dataset size and around 10% faster for 10x dataset size. However, the data extraction stage (colored in blue) is around 84% faster on SEMR. The difference in the data extraction stage between both solutions is larger for 10x dataset size as SEMR executes the data extraction stage around 97% faster than the O-RAN solution. The execution of other overhead tasks and the model deployment stage were not affected by the dataset size. For SEMR, the execution of overhead tasks took around 53s, while for the O-RAN solution, it took around 3s as presented in Figure 16, colored in red. Model deployment (colored in green) required around 52s for SEMR, while the O-RAN solution does not support model deployment so the execution time of this stage was not measured.

Figure 17 presents the average latencies of inference requests sent concurrently to the deployed QoE model. We can see that inference on SEMR and the O-RAN solution perform similarly up to 300 concurrent requests. At 512 concurrent requests, the mean end-to-end latency is 23 seconds for SEMR

and 32 seconds for the O-RAN solution. For larger number of concurrent requests, the SEMR inference has up to 28% lower average end-to-end latency.

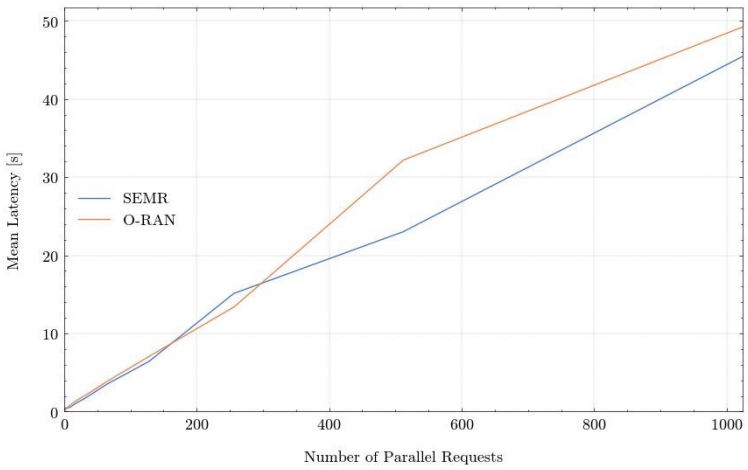

Figure 17. Latency of inference requests

3.4.1. Evaluation of the Triggering Strategies

We evaluate the proactive and reactive triggering strategies of the SEMR using real-world datasets which are generated by the NANCY project. The datasets are collected by JSI[1] and include "*signal strength (RSS) measurements made with Bluetooth Low Energy (BLE) technology, which can be used for outdoor fingerprint-based localization applications*". More specifically, BLE data are collected over a geographical area during the spring and winter seasons, thus two different datasets are generated. In our experiments, we use the data collected in December 2021 as the "old dataset" (named D1) and the data collected in May 2022 as the "new dataset" (named D2). To properly depict the efficiency of our method, we consider the standard deviation difference (STD) between the two datasets.

Proactive approach: Figure 18 illustrates the evaluation of the proactive triggering strategy. We utilize a classification model (MLP), train it with the D1 and then, obtain its F1 score using several variations of the D2. Results demonstrate that as the F1-score drops (due to the increasing STD between the two datasets), the utility score increases. The proactive approach captures this divergence between the two datasets and outputs higher model retraining utility values when the F1-score (and thus, the model quality) drops. As a result, users are prompted to retrain their models, when the dataset changes in a way that cripples the model performance. We should also note that the execution time of the proactive approach is less than 3 seconds, which is considered trivial.

[1] https://zenodo.org/records/7464488#.Y6RMd9WZO3B

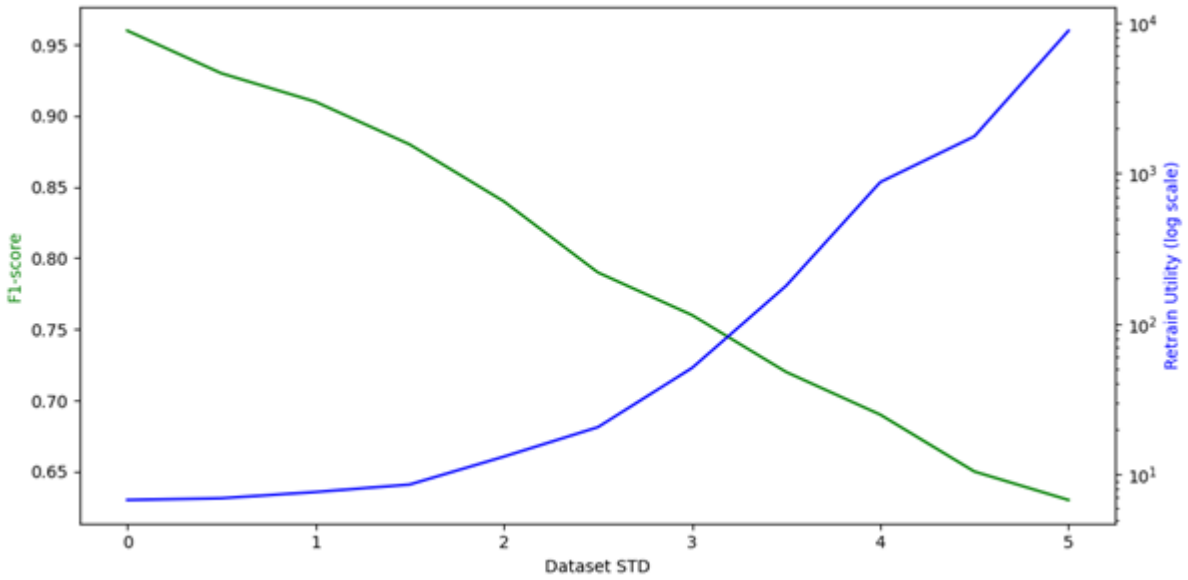

Figure 18. Utility function of the proactive approach over different dataset STD values

Reactive approach: Figure 19 illustrates the evaluation results of the semi-supervised learning approach over different STD values between D1 and D2. Results indicate that the semi-supervised learning approach is very efficient in capturing data divergence. The method is very robust, and the utility value is directly correlated with the STD divergence of the two datasets. As the utility increases, the need for model retraining is more evident, an assumption which is also validated by the increased D1-D2 STD difference. The semi-supervised learning approach requires 120 seconds to run, making it an efficient way to detect data drift.

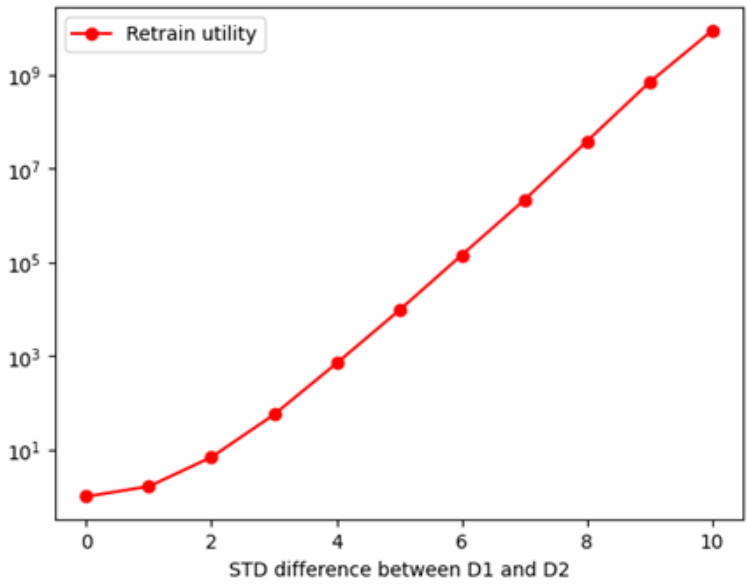

Figure 19. Utility function of the semi-supervised learning method over different dataset STD values

Figure 20 depicts the results we obtained when using the data reconstruction method over the STD difference of the D1 and D2 datasets. The data reconstruction method, which utilizes the TabNet model, manages to efficiently capture the data drift. More specifically, the classification accuracy drops, and the classification loss increases in (almost) a linear way, when data difference is observed. A high loss and low validation accuracy showcases that there is a drift on data and thus, a drift on the model. This output clearly indicates that the data reconstruction method is very efficient in detecting even very small changes in data. The data reconstruction approach requires almost 250 seconds to run, making it ideal for applications which are data sensitive, require high accuracy and have access to better computational resources.

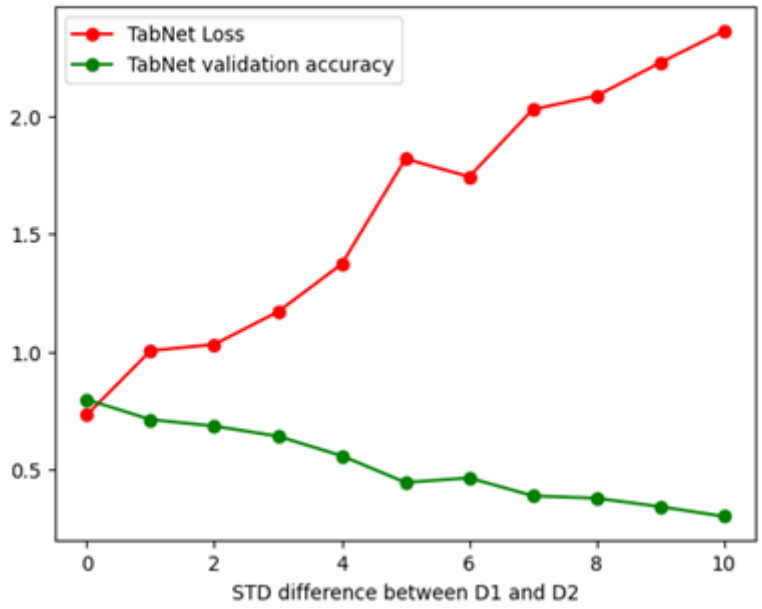

Figure 20. Accuracy and loss of data reconstruction method over different dataset STD values

4. Impact of AI Model Life Cycle on Energy Efficiency

$eCAL$ is a metric proposed to quantify the energy efficiency of an AI model over its entire lifecycle [15]. Unlike traditional metrics such as Energy-per-Bit, which primarily assess the energy efficiency of the communication part of the network, $eCAL$ reveals the additional energy cost for predictions/decisions made by any AI model that is integrated or native to future networks. This comprehensive metric fills the critical gap by capturing energy consumption and number of processed bits from individual components in the network, which includes data collection, storage and data preprocessing, model training and evaluation, and inference. Conceptually, $eCAL$ is similar to the lifecycle emissions in the automobile industry, measured in $[\frac{g}{km}]$ [16], which is employed to compare the emission of electric with traditional vehicles. In the context of 6G networks, where the AI/ML lifecycles are increasingly integrated, systems such as O-RAN [17] are envisioned to manage the AI model lifecycle. Therefore, quantifying the energy cost of AI/ML lifecycle configurations in future networks through the lens of $eCAL$ provides a fundamentally new way to consider sustainability aspects in future network design and configuration.

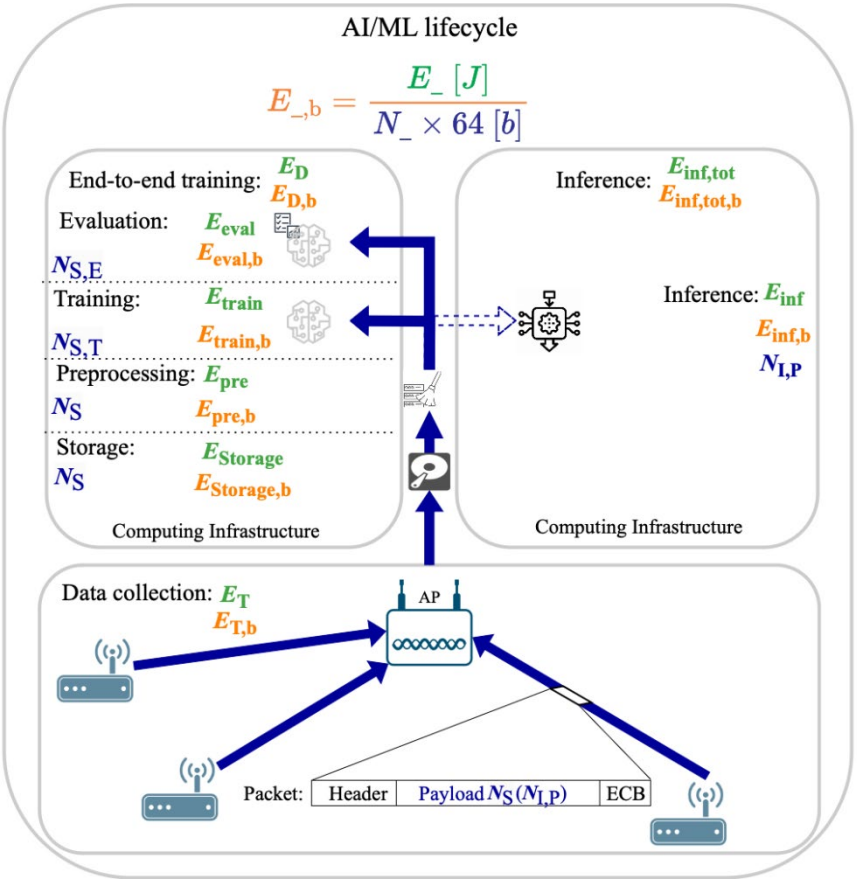

Figure 21. Components in AI/ML lifecycle of wireless communication architecture and their corresponding energy costs and samples.

The AI/ML lifecycle of an AI-powered network can be seen in Figure 21 and consists of the following data manipulating components:

- Data Collection: This component processes the collected samples, denoted as N_S, which includes receiving data from the UE at the computing infrastructure via wired or wireless technologies. For example, the collected information can be transformed into indicators such as signal strength, and further be analysed by the AI models to predict the location of a user

or detect anomalies during transmission. To ensure secure and error-free communication, some overhead bits are added and the total number of transmitted bits is B_T as defined in [15]. The energy required to transmit this data from UE to the Radio Unit (RU) is denoted by E_T [J], as illustrated in Figure 21.

- Storage and Data Preprocessing: Once the data from UE arrives to DU or CU, it is first stored on the hard drive, and then fetched for preprocessing as illustrated in Figure 21. Therefore, the energy consumption, E_{storage}, of this component arises from reading and writing samples N_S into the storage units, which can vary depending on storing technologies and data volume. To ensure accuracy and reliability during the training process, the samples, N_S, must go through several preprocessing steps such as cleaning, feature engineering and transformation. The energy consumption for preprocessing E_{pre}, depends on the integrity of the ingested dataset. For example, datasets with a lot of invalid or missing data, require more extensive cleaning and error correction.
- Training and Evaluation: The training component comprises the model development of the AI/ML using selected AI/ML techniques, such as neural architectures, and relevant data. During this stage, the processed sample, denoted as $N_{S,T}$ is fetched to learn weights and biases in order to approximate the underlying data distribution. In most neural architectures, the learning processes depend on Central Processing Unit (CPU), Graphics Processing Unit (GPU), or Tensor Processing Unit (TPU) performing complex computing operations and consuming E_{train} energy. Once the neural architecture weights are learned using the data in view of minimizing a loss function, the model is considered ready for evaluation and deployment. Subsequently, the quality of the learned model is evaluated on the evaluation datasets, $N_{S,E}$ a process that consumes E_{eval} energy. The relationship between input, training, and evaluation samples is derived in [15], where the authors assume a multilayer perceptron (MLP) model for AI/ML lifecycle , but this can be applied on other architectures as long as the computing complexity can be measured in floating-point operations per cycle per core (FLOPs). As a result, the end-to-end training of AI/ML model, which includes the processes of data storage, preprocessing, training, and evaluation, requires E_D energy to complete.
- Inference: Once the model is trained, it can be utilized by applications in the inference mode. These applications can send samples of data $N_{I,P}$ and receive model outputs in the form of forecasts for regression tasks or discrete predictions for classification tasks. While the energy consumption E_{inf} for a single inference is relatively low, it can become significant for high volumes of requests. Completing an inference requires acquiring data from one UE, followed by data storage and preprocessing. This process is defined in [15] as $E_{\text{inf,p}}$.

Summing the energy consumption and the number of bits processed by each component in the AI/ML lifecycle, we can derive the $eCAL$ of an O-RAN mobile communication system based on the aforementioned AI/ML lifecycle. More specifically, $eCAL$ is defined as the ratio of the energy consumed by all data manipulation components over all the manipulated application-level bits.

$$eCAL = \frac{Total\ energy\ of\ data\ manipulation\ components\ [J]}{Total\ manipulated\ application\ level\ data\ [b]} = \frac{E_D + \gamma E_{\text{inf,p}}}{B_D + \gamma B_{\text{inf,p}}},$$

where γ, B_D, and $B_{\text{inf,p}}$ represent the number of inferences, total bits manipulated in end-to-end training and in a single p-th inference, respectively. Moreover, E_D and $E_{\text{inf,p}}$ are defined as: $E_D = E_T + E_{\text{storage}} + E_{\text{pre}} + E_{\text{train}} + E_{\text{eval}}$

$$E_{\text{inf,p}} = E_T + E_{\text{storage}} + E_{\text{pre}} + E_{\text{inf}}$$

To illustrate the behaviour of $eCAL$, we assume an AI model relies on an MLP with 3 hidden layers and 5 neurons each, whereas on the input layer and output layer there are 6 and 3 neurons, respectively. It requires 256 double-precision samples ($N_S = 256$) from the UE via Bluetooth low energy (BLE) to train and evaluate the model with a 70/30 split. We can then calculate that the AI model requires 404,992 FLOPs for training and 173,568 FLOPs for evaluation. In addition, we utilize HDD for storage (which has an energy consumption of 0.65 $[\frac{Wh}{TB}]$) and normalization as data standardization method (costs $6N_S - 3$ FLOPs). As we observe from the $eCAL$ equation, as the number of inferences increases, the average energy consumption decreases and eventually approaches $E_{\text{inf,p}}$, which is depicted in Figure 22. This observation can also be utilized to measure how well the model is resilient toward performance loss from an energy consumption point of view. More specifically, with the same hardware setup, a model that requires less frequent retraining due to performance degradation will consume less energy than the one that needs frequent retraining.

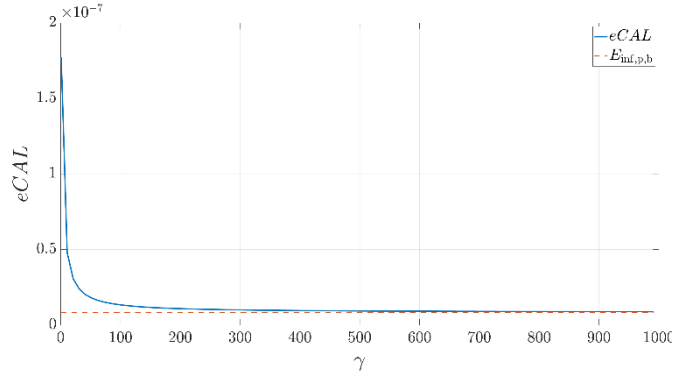

Figure 22. Energy cost of AI/ML lifecycle ($eCAL$) over the number of inferences (γ).

5. Interconnection of AI and Self-Evolving Model Repository

The Self-Evolving Model Repository (SEMR) is a containerized application deployed on Kubernetes. It leverages Helm charts for deployment, with configurations managed through Helm values. The Slice Manager (SM) oversees network and computational slices, ensuring the isolation of containerized network functions (CNFs) and efficient resource allocation for both CNFs and Network Services (NSs). The SM can also deploy SEMR and regulate its resource utilization.

5.1. Requirements

To onboard and instantiate SEMR on a network slice, the following requirements must be met.

1. A computational resource (Kubernetes cluster) must be registered with the Slice Manager (SM) via the SM API.
2. A Compute Chunk and Slice must be created using the SM. This process allocates and isolates the appropriate resources and namespaces within the computational resource for network service instantiation.
3. SEMR's Helm charts need to be onboarded to the SM's Open-Source MANO (OSM).
4. Using the SM's API, network services (in this case, the Self-Evolving Model Repository) can be instantiated and configured using Helm values overrides.

The procedure to satisfy the requirements and deploy SEMR with SM API calls is depicted in Figure 23.

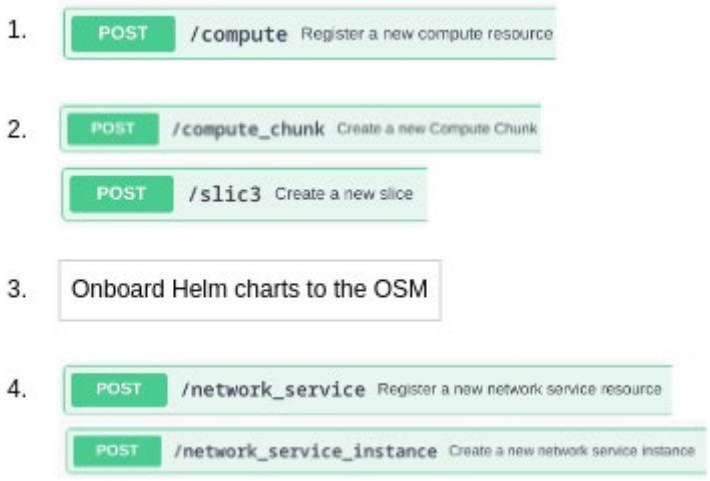

Figure 23. Slice Manager API Calls for Instantiating Network Services

Helm values allow specification of which components of SEMR are enabled, resource allocations for components, and other SEMR component configurations. Helm values are sent in the body of SM POST API requests for network service instantiation. The example Helm values for deployment and instantiation of SEMR are presented in Section 3.3.

5.2. Interconnection Scenario for Model Retraining Initialization

To ensure efficient utilization of compute resources, the SEMR works in an idle state for the majority of the time, as shown in the diagram in Figure 24. Once the SEMR detects the need to retrain the model that it’s orchestrating, it initializes the training process. First, it checks the telemetry data from the Slice Manager to find out the availability of computational resources, after which it requests additional resources to ensure reliable model retraining. Once the model is retrained, the SEMR releases the

additional resources, so that they can be used by other services within the system and returns to the idle state, which requires minimum resources to operate and monitor the lifecycle of orchestrated AI models. This process is fully automated with a DQN reinforcement learning algorithm that controls the Slice Manager through the provided API.

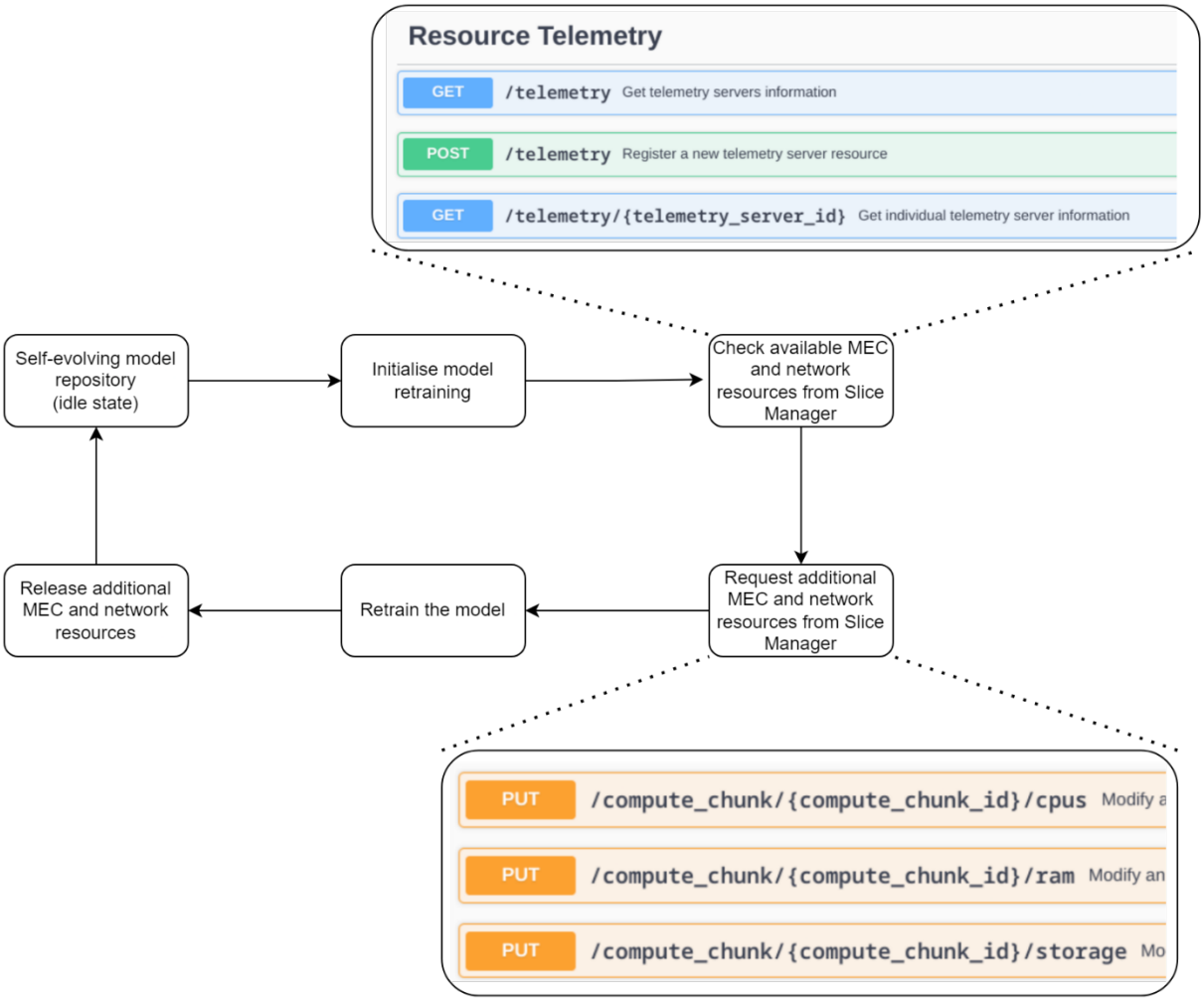

Figure 24 The operational cycle of SEMR and model retraining initialization

6. Workload Scheduling

6.1. Introduction

This section focuses on the scheduling of virtualized workloads on computing nodes by leveraging the SCHED_DEADLINE scheduler to achieve QoS performance and timing isolation. It integrates with the aforementioned AI workflow by leveraging the Ray framework and KubeRay project, which in turn leverages Kubernetes.

Ray is an open-source framework used for building and scaling distributed applications, which is used for the distributed training and inference of AI algorithms. It consists of two key classes of nodes, shown in Figure 25: **(1)** the head node, which is responsible for the coordination, scheduling, and management of a cluster of nodes, and **(2)** a set of Ray worker nodes, which are in charge of performing the actual execution and giving back the results to the head node. In the following, we consider the KubeRay variant of Ray – an open-source project providing native Kubernetes integration for Ray by simplifying the management of Ray clusters using the Kubernetes orchestration features.

In this scenario, the problem of leveraging SCHED_DEADLINE to schedule Ray workloads consists of integrating Kubernetes with SCHED_DEADLINE and it is addressed in the following. The interaction between Ray and Kubernetes is graphically shown in Figure 25. In particular, the Ray Head communicates with the `kube-scheduler`, the Kubernetes component in charge of deciding which worker node to select to deploy a specific container on. In the Kubernetes architecture, each worker node has a dedicated *kubelet*, a component in charge of managing the lifecycle of deployed containers. Each kubelet interacts with a container that includes the Ray worker module corresponding to that node.

Clearly, to leverage the SCHED_DEADLINE scheduler capabilities, the various components need to be modified to consider SCHED_DEADLINE and its key configuration parameters, which are recalled in the following.

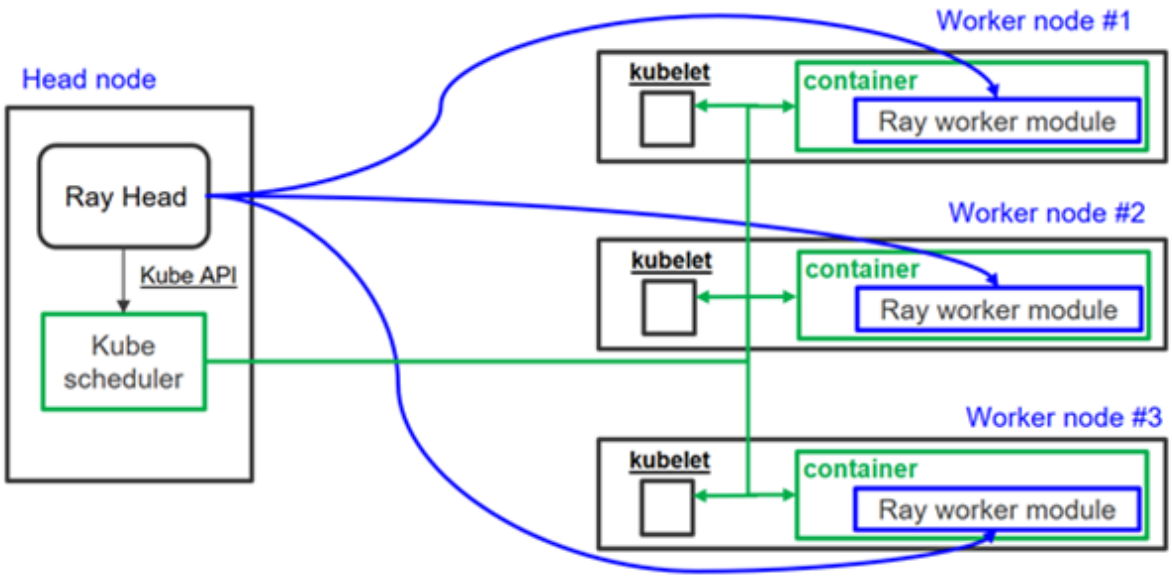

Figure 25. The KubeRay architecture

The SCHED_DEADLINE scheduler of Linux. SCHED_DEADLINE is a scheduler of Linux that allows to provide QoS guarantees and timing isolation to software workloads. It works for general Linux processes, as well as virtualized workloads like QEMU/KVM virtual machines and containers. Containers are made compatible with an out-of-three patch [18]. SCHED_DEADLINE implements the Constant Bandwidth Server (CBS) resource reservation algorithm [19], which allows providing QoS timing guarantees in terms of CPU bandwidth and the worst-case CPU latency for the workload running in the reservation. CBS reservations are scheduled according to the Earliest Deadline First (EDF) scheduling algorithm [19]. SCHED_DEADLINE reservations implement timing isolation through a budgeting mechanism that relies on two key parameters, which need to be configured for each reservation. These parameters are the reservation budget Q (also called runtime) and the reservation period P. In essence, SCHED_DEADLINE provides Q time units of service to the virtualized workload served within the reservation, every period P. SCHED_DEADLINE is both a resource partitioning and a resource enforcement mechanism: it provides a reservation with a fraction of the overall CPU bandwidth (Q/P: this is the resource partitioning part) but it also ensures that it receives no more than the guaranteed fraction (resource enforcement). This way, a faulty, untrustworthy application running in a reservation cannot directly affect the timing behavior of another application due to an overrun caused by a fault or a cyber-attack. Therefore, SCHED_DEADLINE implements a timing isolation mechanism that allows the co-location of different applications on the same processor cores with QoS and temporal isolation guarantees. This allows for saving computational resources that would otherwise be underutilized (e.g., when different services are placed on different cores due to the need to temporally isolate them). These features are also desired in the aforementioned context – nevertheless, the software stacks (e.g., Kubernetes) need to be modified to interact with SCHED_DEADLINE. This effort is presented in the following section.

6.2. Design, Implementation, and Evaluation

First, we focus on making SCHED_DEADLINE compatible with containers, which are used by Ray and Kubernetes, thus obtaining a “Compositional Scheduling Framework” (CSF) [20] [21] [22], which consists of a hierarchy of two schedulers. Indeed, the SCHED_DEADLINE scheduler in mainline Linux allows for encapsulating only one process into a reservation. This is flexible enough for encapsulating regular Linux threads, but also the virtual processors of a KVM/QEMU virtual machine, since each virtual processor is managed as a single process by the host operating system. Differently, managing containers with SCHED_DEADLINE requires including *multiple threads* within the same reservation (associated to a budget and period pair), creating a hierarchy of two schedulers consisting of (1) the scheduling of containers (reservations) according to EDF, and (ii) the scheduling of threads within the reservation, which usually leverages fixed-priority scheduling (via the SCHED_FIFO and SCHED_RR schedulers of Linux, which work in conjunction with SCHED_DEADLINE in this case). In practice, this requires:

1. Modifying the Linux kernel’s CPU scheduler, allowing reservation-based scheduling of *groups* of real-time (SCHED_FIFO or SCHED_RR, both based on fixed-priority scheduling) tasks.
2. Modifying the containerization stack (Kubernetes) to interface it with the new scheduler features mentioned in Item 1.

To schedule real-time tasks in CPU reservations (Item 1), the Linux SCHED_DEADLINE scheduling class has been extended allowing it to schedule control groups (`cgroups`) [18]. In NANCY we extended the “real-time control group scheduling” implementation originally proposed in [18] by improving it to better support massively multi-core servers and porting it to modern kernel versions. Indeed, the

implementation referred in [18] is based on an old version (5.1) of the Linux kernel, while the current scheduler has been massively refactored and contains a new "deadline server" feature that conflicts with the real-time cgroup scheduler, but can be used to simplify its implementation.

Regarding the real-time containerization stack (Item 2), a state-of-the-art prototype [23] (called RT-Kubernetes) showed that it is possible to correctly interface Kubernetes with the real-time cgroup scheduler (this is possible because the real-time cgroup scheduler re-used the "`cpu.rt_…`" interface already used by the vanilla Linux kernel and supported by Kubernetes). Nevertheless, the prototype in [23] only works with a single old version of Kubernetes (in conjunction with an old version of the Linux kernel) and does not address the integration with Ray. The objective of this work is also to make the SCHED_DEADLINE extension to Kubernetes portable across different versions to foster compatibility and maintainability.

The main Kubernetes components that need modifications are the "`kube-scheduler`", which decides the worker node on which a real-time container is going to be scheduled, and the "`kubelet`", which interacts with the container runtime to start the real-time container. These are the components mentioned at the beginning of the section and required to interact with KubeRay.

The kube-scheduler needs modifications because it must ensure that no worker node is overloaded by real-time containers, while the kubelet needs modifications to properly interact with Linux to set SCHED_DEADLINE scheduling attributes, including the budget, period, and number of assigned cores.

The RT-Kubernetes prototype in [23] was invasive and was based on heavyweight modifications of the Kubernetes codebase, that cannot be easily maintained, debugged, or updated. This happened because the prototype did not take advantage of the modular Kubernetes architecture.

Hence, in the context of NANCY, we completely redesigned the RT-Kubernetes, using a more modular, portable and maintainable approach. In particular, the new RT-Kubernetes architecture is based on the Dynamic Resource Allocation (DRA) framework provided by recent Kubernetes versions (see[2]). It is shown in Figure 26.

[2] https://kubernetes.io/docs/concepts/scheduling-eviction/dynamic-resource-allocation/

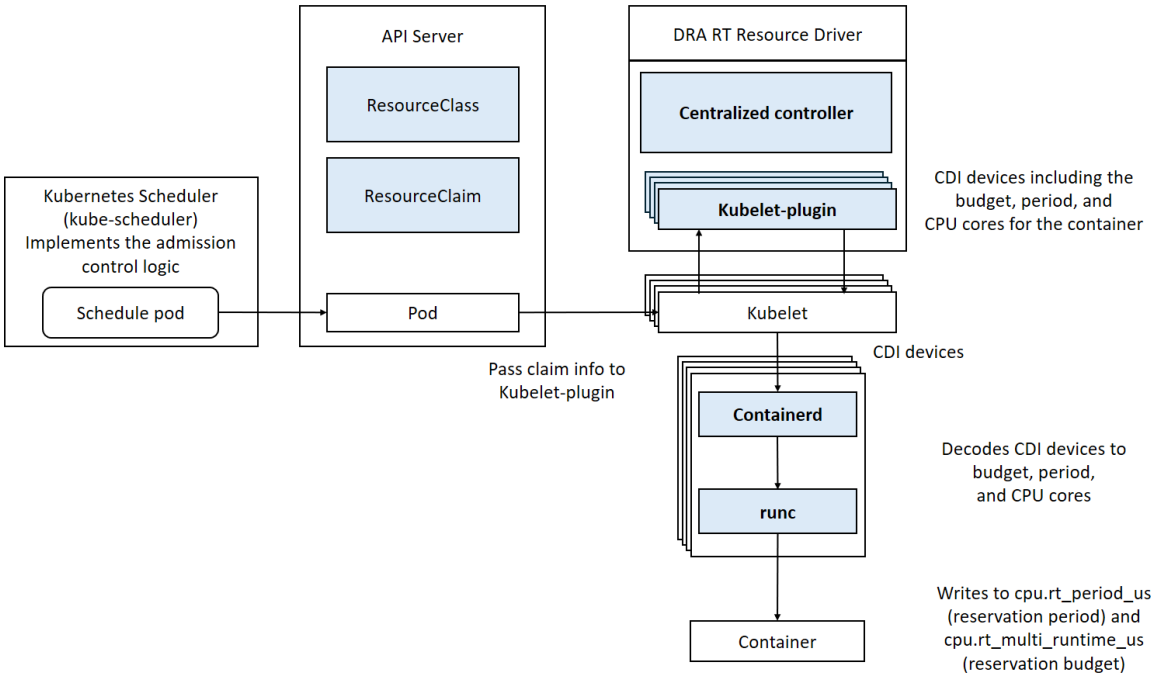

Figure 26. Modular Kubernetes Real-Time Extension

The new feature allows specifying some "*claims*" for each container, describing the container's requirements in terms of system resources, in terms of number of CPU cores, the runtime, and the reservation period for real-time containers. For example, the YAML file describing a (10ms, 100ms, 2) CPU reservation (10ms of execution time every 100ms, on 2 CPU cores) is reported in Figure 27. Then, container specifications can be provided, as in Figure 28, to create real-time containers executing in such a CPU reservation.

```yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: rt-test1

---
apiVersion: resource.k8s.io/v1alpha2
kind: ResourceClaimTemplate
metadata:
  namespace: rt-test1
  name: rt.example.com
spec:
  spec:
    resourceClassName: rt.example.com
    parametersRef:
      apiGroup: rt.resource.example.com
      kind: RtClaimParameters
      name: rtclaims

---
apiVersion: rt.resource.example.com/v1alpha1
kind: RtClaimParameters
metadata:
  namespace: rt-test1
  name: rtclaims
spec:
  count: 2
  runtime: 10000
  period: 100000
```

Figure 27. Definition of a containers using reservations

```
resources:
  claims:
    - name: rtcpu

resourceClaims:
  - name: rtcpu
    source:
      resourceClaimTemplateName: rt.example.com
```

Figure 28. Container definition

The newly developed RT-Kubernetes architecture does not require changes to the kube-scheduler or to the kubelet, but requires to implement a so-called “DRA driver”, which interacts with the various Kubernetes components. The DRA driver is composed of two components: a controller, and a kubelet plugin. When a new pod is created, and Kubernetes needs to start its containers, the kube-scheduler interacts with the DRA driver’s controller to filter the worker nodes that are not suitable to host the new real-time containers. The controller, hence, needs to keep track of the real-time containers running on each worker node of the cluster, and to accept the execution of new real-time containers only on worker nodes that will not be overloaded by the execution of the new containers. In real-time jargon, the controller implements an “admission control” for the worker nodes.

The DRA driver also converts the real-time resource claims into container’s attributes, which are used by the container runtime to set the budget and period on the specified number of CPU cores. This is done by the kubelet plugin by creating a CDI (Container Device Interface) device.

There is a kubelet plugin for each worker node, which registers with the node’s kubelet and receives notifications every time a container is started/stopped on the node.

Currently, this approach requires some small modifications to the container runtime (`runc`, in Figure 26), but such changes are limited and confined to a well-defined part of the code (currently, about 80 lines of “runc” code have been added).

We are currently testing the new RT-Kubernetes implementation and devising ways to remove the need for container runtime modifications, e.g., by setting the cgroups’ scheduling parameters directly in the kubelet plugin.

Figure 29 shows how the execution time of a benchmark application varies with different configurations of the aforementioned parameters. In particular, we show the measured impact of the SCHED_DEADLINE parameters on a virtualized application deployed with the developed system.

When the benchmark executes on an entire processor core, it requires 526ms (dedicated core configuration). This is reported in the last column. The other columns show different execution times that the benchmark can obtain with a different share of the available resources. For example, with only one core assigned and 5% of it assigned (2ms every 40ms), the execution time dramatically increases to 10565ms. However, by shaping the reservation parameters, it is possible to shape also the execution time. For example, 1321ms are required when granting 20% of two cores (8ms every 40ms)

while leaving the rest of the CPU capacity to other applications thanks to the timing isolation capabilities of SCHED_DEADLINE.

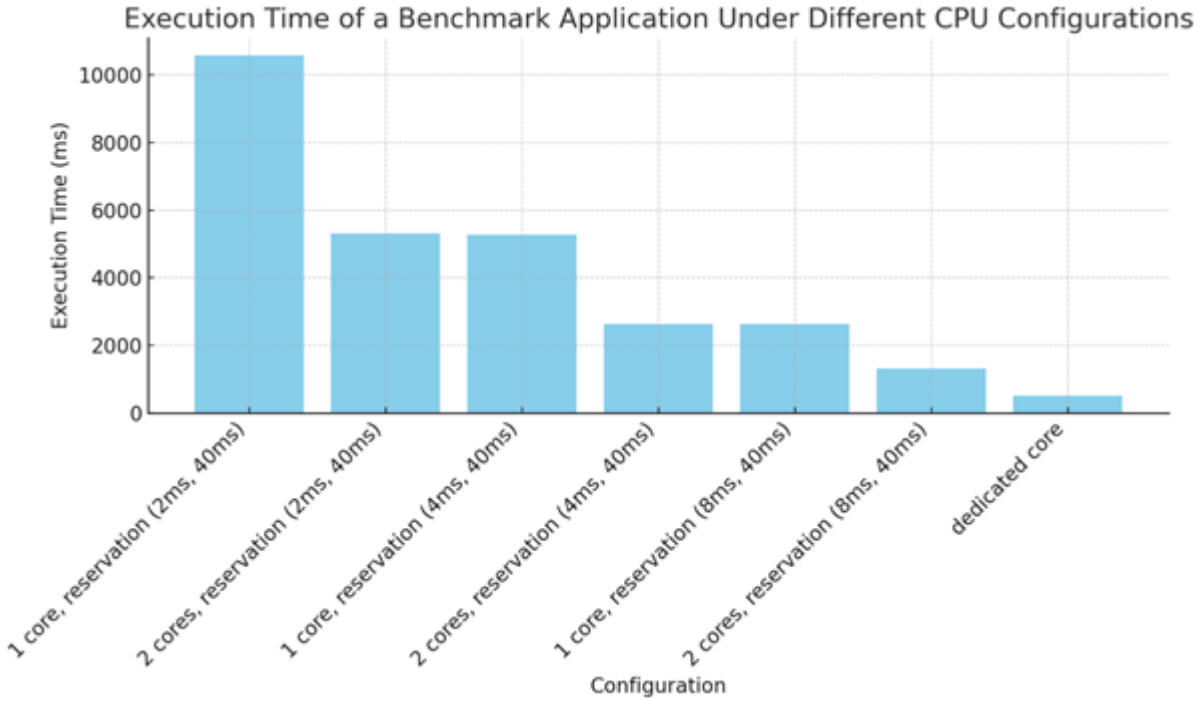

Figure 29. Execution Time of a Benchmark Application Under Different Configurations

7. Integration with the NANCY Platform

7.1. Brief Introduction to the NANCY CI/CD system

The NANCY integrated platform unifies the outcomes of the developed NANCY components (WP2-WP5). To facilitate and optimize the software integration activities, a CI/CD (Continuous Integration/ Continuous Delivery) system has been set up for NANCY. The CI/CD system targets primarily the NANCY containerized components and it consists of a wide range of open-source DevOps tools chosen for the purposes of automating building, testing, and deployment activities. It aims to enhance the reliability of code changes while reducing development cycles.

Continuous Integration (CI) involves developers frequently integrating their code changes into a central repository (NANCY GitHub Organization[3]). Each integration initiates automated builds and a series of tests through a CI server (NANCY Jenkins server[4]) to maintain application stability. For NANCY, this process includes packaging software into Docker container images, storing them in a registry (Harbor container registry[5]), and deploying them in a dedicated development/testing environment, specifically the central NANCY Kubernetes cluster. The primary aim is to accelerate the release cycle by detecting and resolving bugs early, thereby minimizing extensive rework and allowing teams to concentrate more on development and integration.

Continuous Delivery (CD) follows Continuous Integration in the software release pipeline. It involves manually triggering the deployment to "production" environments via the CI server, once all CI workflows have been successfully validated. This phase focuses on preparing the software artifact for distribution to end-users in the NANCY testbeds and demonstration environments.

The NANCY CI/CD system is built on top of Linux Virtual Machines from Hetzner public cloud provider. Figure 30 shows an illustration of the CI/CD system and its connection to the central NANCY Kubernetes cluster that is used for development and testing, as well as the envisioned connection to the different Kubernetes environments of the NANCY testbeds and demonstrators.

The NANCY CI/CD environment and its corresponding open-source DevOps services are summarized in D6.1 and will be presented in detail in D6.2.

[3] https://github.com/NANCY-PROJECT
[4] https://jenkins.nancy.rid-intrasoft.eu
[5] https://harbor.nancy.rid-intrasoft.eu

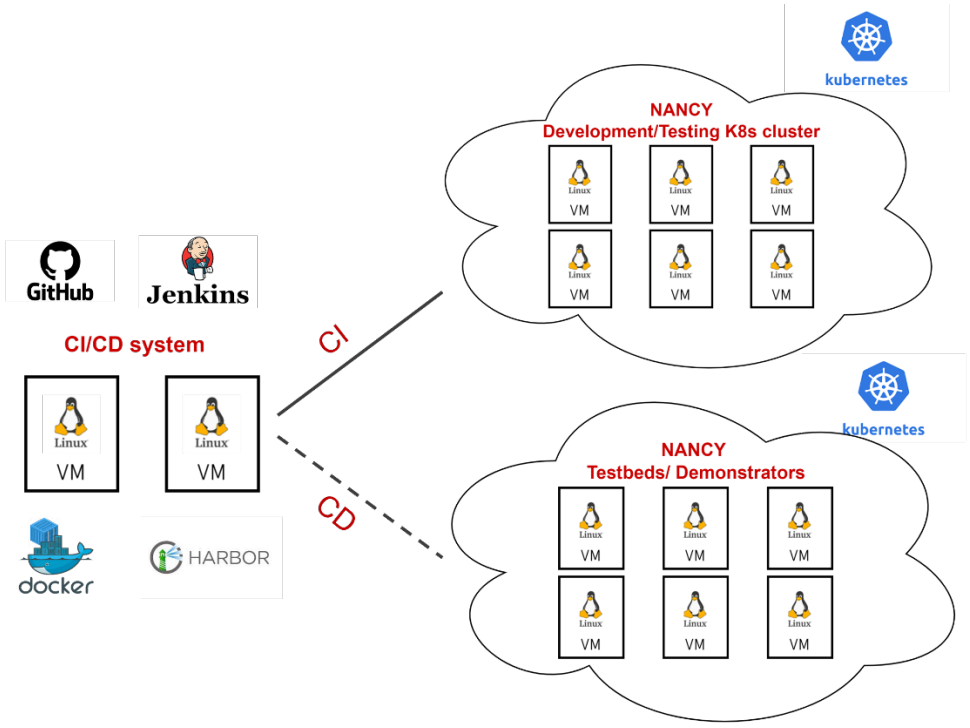

Figure 30: NANCY CI/CD infrastructure and services

7.2. Integration with AI-based B-RAN Orchestration Components

In the context of AI-based B-RAN Orchestration, the NANCY CI/CD services together with the dedicated development/testing environment (Central NANCY Kubernetes cluster) can be used to deploy and test the functionality of SM (section 2) and SEMR (section 3) components, and more importantly their interconnection/ integration (as described in section 5) within a common deployment environment, before their deployment and use in an operational context (NANCY testbeds/demonstrators). For example, as described in section 5.1, we can test the deployment of SEMR through REST API calls to SM that will invoke the corresponding SEMR helm chart with appropriate helm values configuration. Similarly, the resource telemetry API of SM that is invoked from SEMR as described in section 5.2 can also be tested.

Deployments of these components towards the NANCY testbeds and demonstrators will also be leveraged from the CI/CD system services: e.g., NANCY Harbor cloud-native registry supporting container image and helm charts management.

Dedicated workspaces have been created within the NANCY Jenkins CI server and Harbor container registry to accommodate the creation and execution of CI/CD pipelines and hosting of the SM and SEMR components' container images and helm charts, as shown in Figures 31 -34. In this context, role-based access control (RBAC) policies have been applied to restrict access towards the corresponding workspaces to authorized users only.

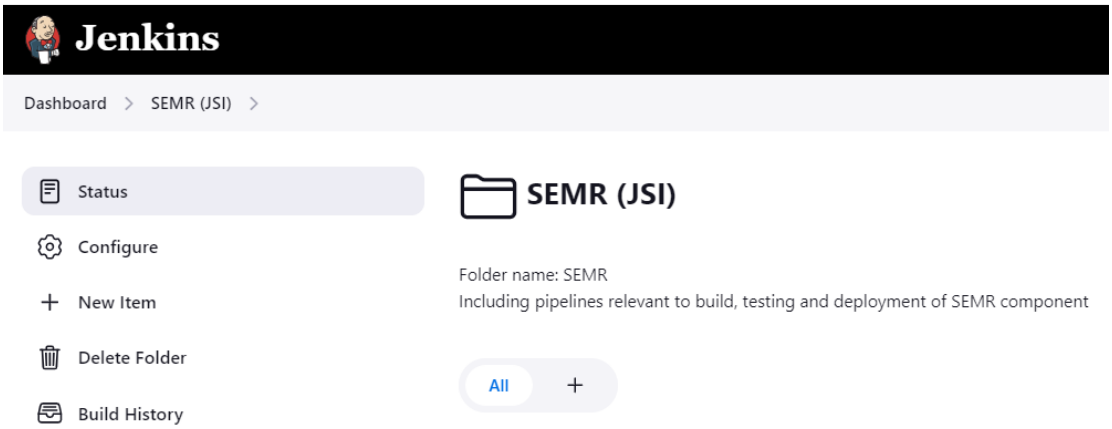

Figure 31: Dedicated Jenkins workspace to configure and manage CI/CD workflows for SEMR component

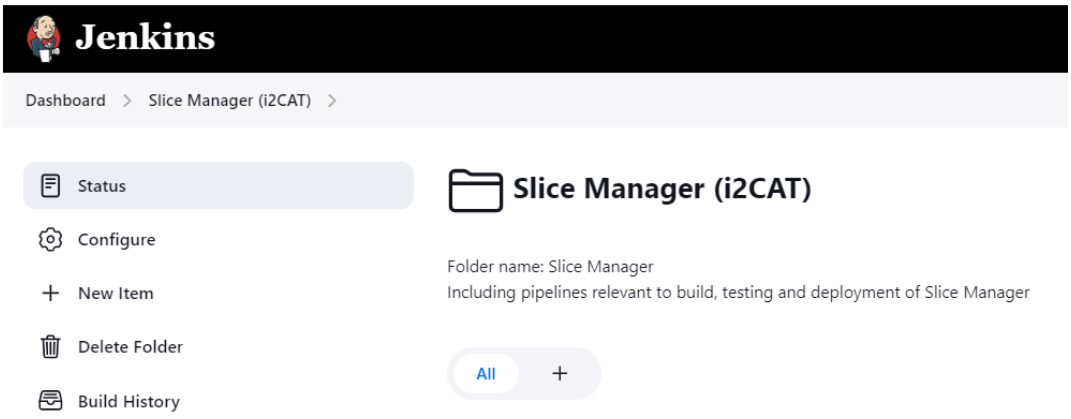

Figure 32: Dedicated Jenkins workspace to configure and manage CI/CD workflows for Slice Manager

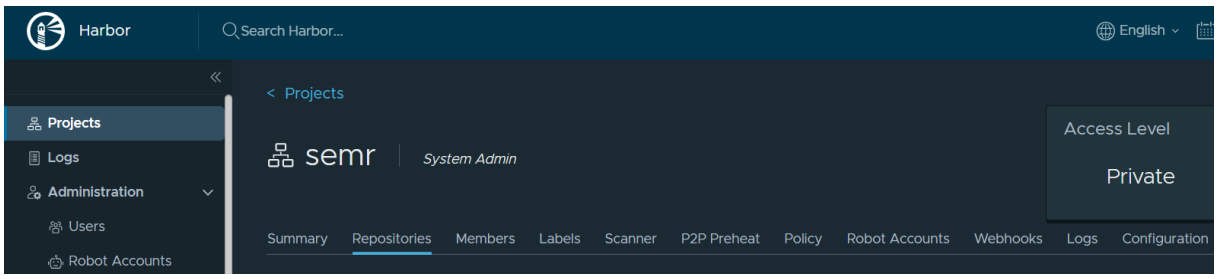

Figure 33: Dedicated Harbor project to host and manage SEMR container images and helm charts

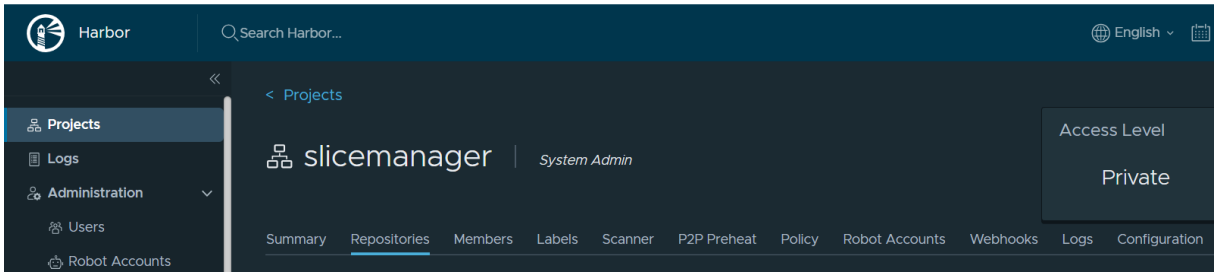

Figure 34: Dedicated Harbor project to host and manage Slice Manager container images

8. Conclusion

This deliverable presents the AI-based B-RAN Orchestration and Self-Evolving Model Repository functionalities along with their interconnection. It quantifies the impact of the AI model on energy efficiency and also proposes a workload scheduler.

A tool for creating and managing network slices in a structured manner is presented. The tool is called Slice Manager and turns out to be a robust and scalable network management solution. Slice creation and activation enable the orchestration of an application's instantiation on top of the created slice. In this context, we introduce the instantiation of services in multi-cluster environment, where the slice reconfiguration takes place according to specific requirements. The position of the Slice Manager is traced in the enforcement block of NANCY architecture on the basis of enabling its actuation on infrastructure by the involved AI decision engines in the network. Next, the Slice Manager implementation is derived. We demonstrate the generation of configuration files for defining configuration parameters, resource limitations and access settings that ensure optimal secure performance. The API Swagger documentation hosting is described, which provides enhanced understanding, testing and consistency in using the API on the running server of the K8 cluster where Slice Manager is deployed and integrating it into applications.

The designed SEMR system allows the automatic modification of the NANCY architecture AI/ML models when the network conditions have changed to such an extent that the new data obtained from the network environment differ drastically from the original data used to train those models. This practice ensures that the models remain relevant and maintain optimal performance despite the continuously changing system parameters. In addition, we specify two elaborate strategies for defining the criteria that will trigger the re-training operation when necessary. Doing so, we ensure that unnecessary training processes are omitted and time, energy and computational resources are more efficiently managed. Finally, the innovative SEMR system, when compared to the conventional ORAN system solution, guarantees significantly smaller workflow execution times and deployment times for the ML algorithms that are incorporated in the NANCY architecture, especially when the dataset size is increasing. This advantage is primarily fueled by the faster data extraction times that more than compensate for its higher model training times. SEMR also exhibits up to 28% smaller average end-to-end latencies of inference requests as their volume scales up.

In the complex NANCY architecture, AI/ML models are constantly incorporated into the system with the objective of optimizing the overall network performance. It is also crucial to quantify the impact of the AI model execution on energy efficiency. To this end, eCAL is proposed as a metric that seeks to quantify the energy efficiency of an AI model over its entire lifecycle. The rationale of the metric is straightforward: the AI/ML model lifecycle is split into four distinct phases/processes, namely data collection, data storage and preprocessing, model training and evaluation, and use in inference mode. The metric then is derived as the ratio of two quantities, the sum of energy consumed in all the model processes over the number of application-level bits that are processed. Thus, eCAL not only promotes sustainability but also scalability. This becomes evident through using a continually increasing number of AI/ML inferences. It is inferred that the less the model is retrained the greater the values that the energy efficiency will assume.

The deployment of SEMR leveraging Helm charts along with required configuration management via Helm values is presented. The requirements that must be met for onboarding and instantiating SEMR as a network service via the SM API are also provided. This allows enabled SEMR component specification as well as component resource allocation and configurations. With regard to detecting the need for model retraining, the SEMR operational cycle is analyzed in detail. This specific model

retraining procedure allows for accurately allocating resources for reliable retraining from SM while releasing additional resources so as to be used by other services within the system.

In the context of NANCY, Workload Scheduling leverages high-end technologies, such as the Linux SCHED_DEADLINE Scheduler and the Kubernetes containerization stack and other components, in order to provide timing isolation and reduce underutilization of the computing and networking resources. The result is a framework, which, above all, views favourably and ensures timing isolation, multiple thread accommodation, satisfactory Quality of Service and container handling via the RT-Kubernetes and DRA driver technologies. Lastly, it is shown that by shaping the reservation parameters, it is possible to effectively control the execution time and leave needed CPU capacity to other applications thanks to the timing isolation capabilities of SCHED_DEADLINE.

As far as integration goes, NANCY has been set to encompass and include a great deal of diverse containerized software components. This is achieved via the CI/CD process, operationalized through the use of the Jenkins dedicated workspace, Harbor, Slack and, mainly, GitHub. Although the seamless interoperability of the NANCY elements is expectedly a great challenge, given their massive complexity and diversity, in NANCY we accomplish it, through employing a NANCY-specific Kubernetes cluster and taking advantage of its rich software component integration capabilities.

Bibliography

[1] R. Su, D. Zhang, R. Venkatesan, Z. Gong, C. Li, F. Ding, F. Jiang and Z. Zhu, " Resource Allocation for Network Slicing in 5G Telecommunication Networks: A Survey of Principles and Models," *IEEE Network,* vol. 33, no. 6, pp. 172-179, 2019.

[2] A. Ksentini and P. Frangoudis, "Toward Slicing-Enabled Multi-Access Edge Computing in 5G," *IEEE Network,* vol. 34, no. 2, pp. 99-105, 2020.

[3] D. Kreuzberger, N. Kühl and H. S, "Machine Learning Operations (MLOps): Overview, Definition, and Architecture," *IEEE Access,* vol. 11, pp. 31866-31879, 2023.

[4] I. Sorkhoh, D. Ebrahimi, R. Atallah and C. Assi, "Workload Scheduling in Vehicular Networks With Edge Cloud Capabilities," *IEEE Transactions on Vehicular Technology,* vol. 68, no. 9, pp. 8472-8496, 2019.

[5] R. Lin, X. Guo, S. Luo, Y. Xiao, B. Moran and Z. M, "Application-aware computation offloading in edge computing networks," *Future Generation Computer Systems,* vol. 146, pp. 86-97, 2023.

[6] J. Diaz-de-Arcaya, A. I. Torre-Bastida, G. Zárate, R. Miñón and A. Almeida., "A Joint Study of the Challenges, Opportunities, and Roadmap of MLOps and AIOps: A Systematic Survey," *ACM Computing Surveys,* 2023.

[7] C. Dellarocas, M. Klein and H. Shrobe, "An architecture for constructing self-evolving software systems," in *Proceedings of the third international workshop on Software architecture*, 1998.

[8] O.-R. S. Community, "O-RAN SC Documentation," 2024. [Online]. Available: https://o-ran-sc.readthedocs.io.

[9] O.-R. Alliance, "O-RAN Working Group 2 AI/ML workflow description and requirements, ORAN-WG2. AIML.," 2021.

[10] L. Peterson, T. Anderson, S. Katti, N. McKeown, G. Parulkar, J. Rexford, M. Satyanarayanan, O. Sunay and A. Vahdat, "Democratizing the Network Edge," *ACM SIGCOMM Computer Communication Review,* vol. 49, no. 2, pp. 31-36, 2019.

[11] A. Ibrahim, T. Tarik, S. Konstantinos, K. Adlen and F. Hannu, "Network Slicing and Softwarization: A Survey on Principles, Enabling Technologies, and Solutions," *IEEE Communications Surveys & Tutorials,* vol. 20, no. 3, pp. 2429-2453, 2018.

[12] T. Tarik, S. Konstantinos, M. Badr, F. Hannu, D. Sunny and S. Dario, "On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration," *IEEE Communications Surveys & Tutorials,* vol. 19, no. 3, pp. 1657-1681, 2017.

[13] A. O. Sercan and T. Pfister, "TabNet: Attentive Interpretable Tabular Learning," *Proceedings of the AAAI Conference on Artificial Intelligence,* vol. 35, pp. 6679-6687, 2021.

[14] "AI/ML Framework," [Online]. Available: https://lf-o-ran-sc.atlassian.net/wiki/spaces/AIMLFEW/overview?pageId=13697038. [Accessed 12 2024].

[15] S.-K. Chou, J. Hribar, M. Mohorcic and C. Fortuna, "The energy cost of artificial intelligence of things lifecycle," *Arxiv,* 2024.

[16] J. Buberger, A. Kersten, M. Kuder, R. Eckerle, T. Weyh and T. Thiringer, "Total CO2-equivalent life-cycle emissions from commercially available passenger cars," *Renewable and Sustainable Energy Reviews,* vol. 159, pp. 112-158, 2022.

[17] A. Giannopoulos, S. Spantideas, N. Kapsalis, P. Gkonis, L. Sarakis, C. Capsalis, M. Vecchio and P. Trakadas, "Supporting intelligence in disaggregated open radio access networks: Architectural principles, AI/ML workflow, and use cases," *IEEE Access,* vol. 10, pp. 39580-39595, 11 April 2022.

[18] L. Abeni, A. Balsini and T. Cucinotta, "Container-based real-time scheduling in the Linux kernel," *ACM SIGBED Review,* vol. 16, no. 3, pp. 33-38, 2019.

[19] L. Abeni, G. Lipari and J. Lelli, "Constant bandwidth server revisited," *ACM SIGBED Review,* vol. 11, no. 4, pp. 19-24, 2015.

[20] I. Shin and I. Lee, "Compositional real-time scheduling framework," *25th IEEE International Real-Time Systems Symposium,* pp. 57-67, 2004.

[21] I. Shin and I. Lee, "Compositional Real-time Scheduling Framework with Periodic Model," *ACM Transactions on Embedded Computing Systems,* vol. 7, no. 3, pp. 1-39, 2008.

[22] E. Bini and M. Bertogna, "ACM Transactions on Embedded Computing Systems," *30th IEEE Real-Time Systems Symposium,* pp. 437-446, 2009.

[23] S. Fiori, L. Abeni and T. Cucinotta, "RT-kubernetes: containerized real-time cloud computing," *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing ,* pp. 36-39, 2022.