NANCY

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

Deliverable 3.4

NANCY AI Virtualiser

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 Virtualiser
Deliverable Number	D3.4
Work Package	WP3
Associated Task	T3.4 AI virtualiser for underutilized computational & communication resource exploitation
Dissemination Level	Public
Due Date	M24
Completion Date	23 December 2024
Submission Date	24 December 2024
Deliverable Lead Partner	VOS
Deliverable Author(s)	Daniel Raho (VOS), Alvise Rigo (VOS), Anna Panagopoulou (VOS), Vasileios Anagnostoulis (VOS), Edoardo Manfredi (VOS), Hatim Chergui (i2CAT), Luca Abeni (SSS), Daniel Casini (SSS), Mauro Marinoni (SSS), Alessandro Biondi (SSS), Ramon Sanchez Iborra (UMU), Gonzalo Alarconh (UMU), Rodrigo Asensio Garriga (UMU), Panagiotis 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	30/04/2024	Initial outline	Alvise Rigo, Anna Panagopoulou	VOS
0.2	21/03/2024	Added input to Section 5	Gonzalo Alarconh	UMU
0.3	02/09/2024	Section 1	Alvise Rigo, Anna Panagopoulou	VOS
0.4	12/09/2024	Added input to Section 4	Luca Abeni, Daniel Casini, Mauro Marinoni, Alessandro Biondi	SSS
0.5	15/09/2024	Added input to Section 2 and Section 3	Hatim Chergui, Alvise Rigo, Anna Panagopoulou, Vasileios Anagnostoulis, Edoardo Manfredi, Konstantinos Kyranou, Georgios Niotis, Georgios Michoulis, Georgios Tziolas	I2CAT, VOS, SID
0.6	17/09/2024	Added input to Section 5	Rodrigo Asensio Garriga	UMU

0.7	23/09/2024	Added input to Section 6.2 and Section 3	Anna Panagopoulou	VOS
0.8	27/09/2024	Added input to Section 7	Panagiotis Matzakos	INTRA
0.9	28/09/2024	Added input to Section 6.1	Hatim Chergui	I2CAT
0.99	15/10/2024	Prepare for Quality Check	Alvise Rigo, Anna Panagopoulou, Daniel Casini, Ramon Sanchez Iborra	VOS, SSS, UMU
1.0	05/12/2024	Address reviewers' remarks	Alvise Rigo, Anna Panagopoulou, Alessandro Biondi, Hatim Chergui	VOS, SSS, I2CAT

Internal Review History

Name	Organisation	Date
Stylianos Trevlakis, Lamprini Mitsiou	INNO	22 October 2024
Blaz Bertalanic	IJS	04 November 2024

Quality Manager Revision

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

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Explanation
B5G	Beyond Fifth-Generation
AI	Artificial Intelligence
ML	Machine Learning
NG-SDN	Next-Generation Software Defined Networking
NFV	Network Functions Virtualization
CI/CD	Continuous Integration and Continuous Delivery/Deployment
DRL	Deep Reinforcement Learning
MA-DRL	Multi-Agent Deep Reinforcement Learning
eMMB	Enhanced Mobile Broadband
uRLLC	Ultra Reliable Low Latency Communications
mMTC	Massive Machine-Type Communications
CPU	Central Processing Unit
vRAN	Virtual Radio Access Network
O-RAN	Open Radio Access Network
CU / DU	Central Unit / Distributed Unit
RRM	Radio Resource Management
RLC	Radio Link Control
PRB	Physical Resource Block
DQN	Deep Q-Network
IB	Information Bottleneck
DNN	Deep Neural Networks
IQR	Interquartile Range
SM	Slice Manager
API	Application Programming Interface
VNF	Virtual Network Functions
NFVO	Network Functions Virtualisation Orchestrator
VNFM	Virtualised Network Function Manager
VIM	Virtualised Infrastructure Manager
NFVI	Network Functions Virtualisation Infrastructure
IaaS	Infrastructure-as-a-Service
KPI	Key Performance Indicator
MEC	Multi-access Edge Computing
ARM	Advanced RISC Machine
VM	Virtual Machine
ETSI	European Telecommunications Standards Institute
NFV MANO	NFV Management and Orchestration
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
URI	Uniform Resource Identifier
EL	Exception Level
SMC	Secure Monitor Call
PSCI	Power State Coordination Interface
PMU	Power Management Unit
QoS	Quality of Service
SLA	Service Level Agreement
E2E	End-to-end
DoS	Denial of Service

Executive summary

This document presents the developed solutions in the context of the AI Virtualiser component of NANCY. These solutions share the common objective of optimizing the resource utilization and concern various layers of the networked infrastructure, which range from the Slice Manager layer to the edge.

First, the deliverable describes in detail the exploitation of the under-utilized resources at the Slice Manager layer (Section 2), including a throughout presentation of the applied MA-DRL protocol learning mechanism (Section 2.1) as well as the containerization of the components at this layer (Section 2.2). Second, the deliverable presents the optimized exploitation of resources at the Edge level (Section 3), focusing on a novel virtualisation solution that aims to maximally exploit the computational capabilities at the edge. The document focuses on the integration of the virtualization solution with high-level network orchestrators (Section 3.1) and describes in detail the novel hypervisor driver along with its internal functionalities (Section 3.2). Then, the deliverable presents the mechanisms of the AI Virtualiser that are developed to realize the optimal provisioning of CPU resources through fine-grained decisions at an inter-domain layer (Section 4). Specifically, Section 4.1 introduces in detail the problem of under-utilizing in Linux the virtualized computational resources while Section 4.2 elaborates on the novel mechanism for runtime monitoring of the virtualized CPUs. For the aforementioned technologies code snippets and experimental results are also presented.

Next, the deliverable presents the placement of the AI Virtualiser components into NANCY functional and deployment architecture (Section 5), as part of the overall integration. Additionally, it also presents the association of the AI Virtualiser with the Task Offloading module of NANCY (Section 5.1), as they are closely connected for ensuring the QoS in the context of an agreed SLA. Then, the deliverable concludes with a description of the AI Virtualiser integration in NANCY's testbeds and demonstrators (Section 6) and the relevant CI/CD environment setup (Section 7).

1. Introduction

1.1. Purpose of the Document

As 5G networks and beyond continue to evolve, they bring with them a great deal of complexity and a transformative potential for a wide range of industries. These networks are designed to support a massive number of connected devices and accommodate highly-diverse service requirements, including reliable and ultra-low-latency executions. In these regards, legacy network orchestrators are no longer suitable for effectively managing B5G networks, especially in light of the newly-emerging and demanding applications. Instead, 5G necessitates the development of advanced orchestrator tools that are able to manage resources, optimize network slices, and ensure end-to-end service quality in real time.

On the other hand, AI and ML techniques have rapidly evolved, especially as networks have become more complex and demanding. These advancements present new opportunities for enhanced network orchestration, making it possible to integrate intelligent modules that can better manage the dynamic nature of modern networks. Thus, we have now the opportunity to empower the state-of-the-art orchestrators with intelligent modules, so that they can make more informed decisions to proactively and reactively adapt to network changes. Integrating with pioneering AI and ML techniques, the new orchestrators aim to significantly improve the experience of users as well as contribute to scaling up the overall network performance.

In this context, NANCY introduces the AI Virtualiser as a central component of the architecture that enriches the orchestrator module with the incorporation of cutting-edge technologies and intelligent ML algorithms. Having as target to effectively identify and maximally exploit the available network resources, these technologies share a common objective: To help optimize the resource utilisation.

The AI Virtualiser is a multi-layered solution which extends from the Slice Manager layer to edge-focused innovations. The key asset of the AI Virtualiser lies in this combination of intelligent solutions close to the orchestration modules and finer-grained solutions at the edge, all powered by a suite of modern 5G technologies such as Slicing, NG-SDN and NFV.

At the Slice Manager layer, the AI Virtualiser introduces novel Reinforcement Learning algorithms, aiming to achieve intelligent adjustments of the network slices by rewarding these configurations that effectively minimize the under-utilisation of the available resources. Through the dynamic re-configuration of slices, the AI Virtualiser eventually adapts to the changing network conditions and gradually learns to provide the most efficient, in terms of resource utilisation and performance, slice deployments.

At the same time, the AI Virtualiser establishes innovative NFV-based Virtualization techniques tailored to the characteristics of powerful devices at the network edge, with the dual purpose of exploiting to the maximum extent the available computation resources at this layer and achieving outstanding execution performance for the Virtualized Functions.

Last but not least, the AI Virtualiser seeks to minimize the under-utilisation of the processing bandwidth of the network, as it incorporates new tools to achieve fine-grained granularity in the allocation decisions of the processing resources. Specifically, this is made possible through the introduction of intelligent and dynamic configurations at the scheduling layer of the Operating Systems that are deployed on the compute-capable network nodes.

This document presents the NANCY's AI Virtualiser component, built to address important resource utilisation challenges in the context of the continuously evolving B5G networks. This deliverable reports the activities carried out under the scope of Task 3.4: "AI Virtualiser for underutilized computational & communication resource exploitation", which achieves the result R8. In the pages that follow, the end-to-end architecture of the AI Virtualiser is thoroughly presented and its constituents are individually detailed. Then, the interactions between the AI Virtualiser to other components of the NANCY architecture are described, zooming specifically on the "Computational Offloading & User-centric Caching Functionalities" outcomes of Task 4.1. Finally, the deployment of the AI Virtualiser technologies in the scope of specific NANCY use-cases is explained, along with the instantiated testbed environments. What is more, the CI/CD pipeline implementation for the testing of the AI Virtualiser is described in detail. Finally, the document concludes with a summary of the key insights and takeaways.

1.2. Relation with other Tasks and Deliverables

As anticipated above, this deliverable collects the results of the Task 3.4 activities, detailing the problems and challenges the task had to address and the strategies and technologies devised. Task T3.4 received the NANCY Requirements Analysis (D2.1) and the NANCY Architecture Design (D3.1) in input.

Furthermore, this deliverable connects to Task 4.1 as, both the AI Virtualiser and the Computational Offloading & User-centric Caching Functionalities, jointly, aim to improve the efficiency of the NANCY architecture as a whole.

1.3. Structure of the Document

The rest of the document is structured as follows:

- **Section 2 – AI Virtualiser and Slice-level Resource Exploitation** presents the technologies implemented by the AI Virtualiser at the slice manager layer as a means to optimize the slice-level resources exploitation.
- **Section 3 – AI Virtualiser and Edge-level Resource Exploitation** describes the mechanism implemented by the AI Virtualiser to better exploit through novel virtualization technologies the resources present at the edge.
- **Section 4 – AI Virtualiser and Inter-domain CPU Resource Provisioning** documents the techniques used inside the compute-capable nodes to optimise the utilisation of CPU resources thanks to ad-hoc scheduling technologies.
- **Section 5 – AI Virtualiser on the NANCY Functional and Deployment View** presents the placement of the AI Virtualiser components as part of the overall NANCY functional and deployment architecture. This section also outlines the association between the AI Virtualiser and offloading capabilities.
- **Section 6 – AI Virtualiser in NANCY Testbeds and Demonstrators** describes the testing and validation deployments expected for the AI Virtualiser components.
- **Section 7 – CI/CD Integration of the AI Virtualiser and Testing** is focused on detailing the AI Virtualiser CI/CD infrastructure and connected testing strategies.
- **Section 8 – Conclusion** concludes the deliverable.

2. AI Virtualiser and Slice-level Resource Exploitation

2.1. MA-DRL protocol learning

2.1.1. Protocol Learning for Minimizing Inter-Slice Resource Underutilisation and Conflicts

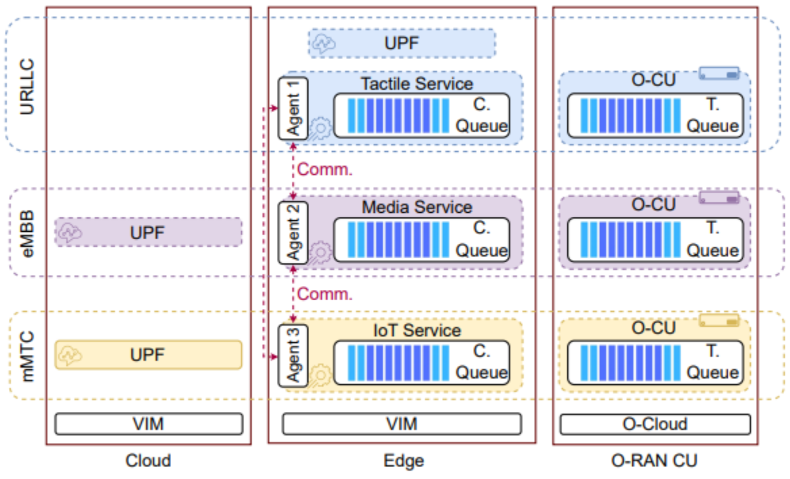


Figure 1: Architecture of the inter-slice conflict resolution use case, with one agent per slice, where the O-RAN and Edge domains form the network slicing environment

Experimental evaluations [1] reveal a non-linear relationship between bitrate, virtualized RAN (vRAN) bandwidth, and CPU utilisation, highlighting that even with adequate radio resources, network performance can degrade without sufficient computational resources in the O-Cloud. Effective radio resource management (RRM) in O-RAN's vRAN setup requires strategic and coordinated CPU allocation. In the scenario depicted in Figure 1, an inter-slice intelligent resource orchestration use-case is presented, where each slice consists of a server located at the edge domain, managed by the virtual infrastructure manager (VIM) responsible for computing, storage, and network resources. The VIM operates within a cloud environment and handles a *Computation Queue* with preemptive CPU resources, affecting local latency. The O-RAN domain includes a per-slice *Transmission Queue* at the O-CU level, where latency depends on radio conditions. Excluding x-haul delays, slice latency is a combination of O-RAN and edge latencies. Each slice-level resource orchestration agent can dynamically adjust CPU frequency and utilize additional resources if other slices leave them underutilized. Given the limitations of centralized approaches, agents collaborate via a decentralized multi-agent deep reinforcement learning (DRL) framework, coordinating through the exchange of actions or messages. These DRL agents must jointly learn the signaling policy without a pre-defined agreement on control messages, guided by a reward function that penalizes conflicts and underutilisation as well as minimizes latency. By properly scaling the CPU frequency, the agent minimizes the computation queue delay and therefore the latency, while the messages that are guided by the reward function avoid conflicts and underutilisation.

2.1.2. Environment's Architecture and Experiments

We consider a scenario with service-specific slices represented by data sourced from a simulator, capturing the nuances of services like eMBB, URLLC, and mMTC in terms of traffic and radio resources. The simulator comprises three core components: physical (PHY), MAC, and radio link control (RLC) functions, and includes the O-RAN E2 interface for collecting network statistics from each distributed unit (O-DU). To prevent underutilisation, each slice has a share of CPU resources that can also be utilized via preemption by other slices if no conflict arises. Each slice is equipped with the AI virtualizer agent that aims to optimize CPU frequency allocation and resolve inter-slice resource conflicts through a conflict resolution protocol.

The default CPU allocation is defined by the array [15, 15, 10], representing 15 GHz for the first and second slices and 10 GHz for the third slice. This approach can also apply to other resource allocation problems, such as PRB allocation. Each agent employs a DQN incorporating an Information Bottleneck (IB) block with a stochastic bottleneck layer, which captures essential features while discarding redundant information. A prioritized experience replay is used to sample experiences based on importance, focusing on pivotal network states to ensure generalization across scenarios.

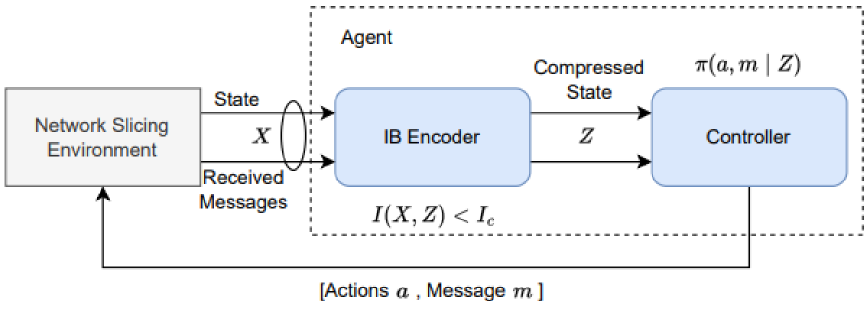


Figure 2: Agent architecture and interfaces

To ensure generalizability, a time-series and variability analysis was conducted on the traffic across three slices. The analysis provides insights into the AI Virtualiser's adaptability to varying traffic conditions, quantified by different standard deviations. For eMBB Traffic, the mean is 23.33 Mbps with a standard deviation of 22.38 Mbps. For URLLC Traffic, the mean is 7.80 Mbps with a standard deviation of 9.25 Mbps. For mMTC Traffic, the mean is 14.80 Mbps with a standard deviation of 20.86 Mbps.

The designed network slicing environment simulates the interactions among agents and their impact on the overall network performance. As agents optimize CPU frequencies and send messages, the environment updates the system state and returns rewards, designed to penalize conflicts and reward low latency. The reward function includes an exponential component to increase rewards as latency decreases and penalizes heavily when resource constraints are violated. During training, the agents use an ϵ\epsilonϵ-greedy strategy for exploration, with ϵ\epsilonϵ decaying over episodes.

Experiments were conducted on a server with two Intel(R) Xeon(R) Gold 5218 CPUs @ 2.30GHz and dual NVIDIA GeForce RTX 2080 Ti GPUs. The DNN network comprises an input layer mapped to a 64-neuron hidden layer activated by the ReLU function, followed by an IB layer with a bottleneck dimension of 32. The network balances the compression and preservation of relevant information using a parameter β that adjusts the trade-off within the loss function. A fully connected layer outputs both main actions and messages. During training, weights are updated using the Adam optimizer with a learning rate of 0.0005. The training process runs for 500 episodes with a maximum of 60 steps per episode, using a batch size of 64, a discount factor γ of 0.99, and an initial β of 0.0 that anneals at a rate of 0.001 per episode. This simulation, developed with libraries like torch and gym, validates resource allocation in a network slicing scenario and optimizes service-specific slices.

2.1.3. Evaluation Results

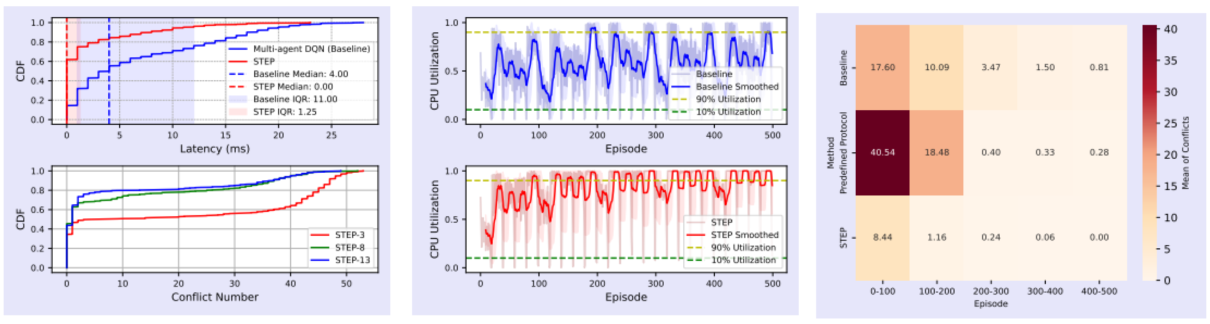


Figure 3: Evaluation Results

Figure 3 presents the evaluation results focusingon the AI virtualizer approach compared to a baseline Multi-Agent Deep Reinforcement Learning (MADRL) framework and a predefined protocol method in a network slicing environment [2]. Here is a summary of key findings:

- **CPU Utilisation/Exploitation Efficiency:** The AI Virtualiser (denoted STEP in Figure 3) significantly improved CPU utilisation compared to the baseline. By leveraging an emergent protocol, agents could temporarily use free CPU resources from other slices, thereby enhancing overall network-wide utilisation. The AI Virtualiser surpassed the lower and upper utilisation thresholds of 10% and 90%, respectively, achieving up to 1.4 times better CPU utilisation than the baseline, which struggled with consistent performance.
- **Resource Conflict Resolution:** During the initial 100 episodes of training, the baseline MADRL framework averaged 17.6 conflicts between slices, while the AI Virtualiser approach showed significantly fewer conflicts, averaging 8.44. The predefined protocol approach, where agents communicated their resource allocation strategies using three distinct codes (0, 1, and 2), initially averaged 40.54 conflicts. Over time, the AI Virtualiser approach consistently outperformed the baseline, eventually reducing conflicts to zero by the final set of 100 episodes. In comparison, the baseline and predefined protocol methods averaged 0.81 and 0.28 conflicts, respectively. Overall, the AI Virtualiser reduced inter-slice conflicts by a factor of 3.4 and 6.06 compared to the baseline and predefined protocol methods, highlighting the effectiveness of adaptive communication in resource allocation.
- **Latency Performance:** The AI Virtualiser framework demonstrated superior performance in reducing average transmission and computation latencies compared to the baseline. The cumulative distribution function (CDF) plot showed the baseline's median latency at 4 ms, while the AI Virtualiser achieved a median latency of 0 ms. The interquartile range (IQR) further highlighted this difference: the baseline's IQR was 11 ms, indicating a broader spread in latency values, whereas the AI Virtualiser 's IQR was only 1.25 ms, signifying minimal variability. Overall, the AI Virtualiser achieved an average latency reduction of 3.5 times compared to MADRL, showing consistent and low-latency performance across slices.
- **Impact of Communication Space Size on Conflicts:** An analysis of conflicts with varied communication space sizes (i.e., different message sets exchanged between agents) revealed that while increasing message diversity initially improves inter-agent communication, it eventually reaches a saturation point. Excessive communication could result in a homogeneous strategy where slices handle similar tasks, reducing diversity in conflict resolution approaches. The tests, conducted under stringent conditions, demonstrated that effective communication must strike a balance to optimize resource allocation.

In conclusion, the AI Virtualiser strategy proved highly effective in optimizing resource allocation, reducing conflicts, and improving latency and CPU utilisation. Its adaptive communication capabilities allow agents to learn efficient protocols without predefined rules, resulting in superior performance compared to conventional approaches.

2.2. Containerization of the Components

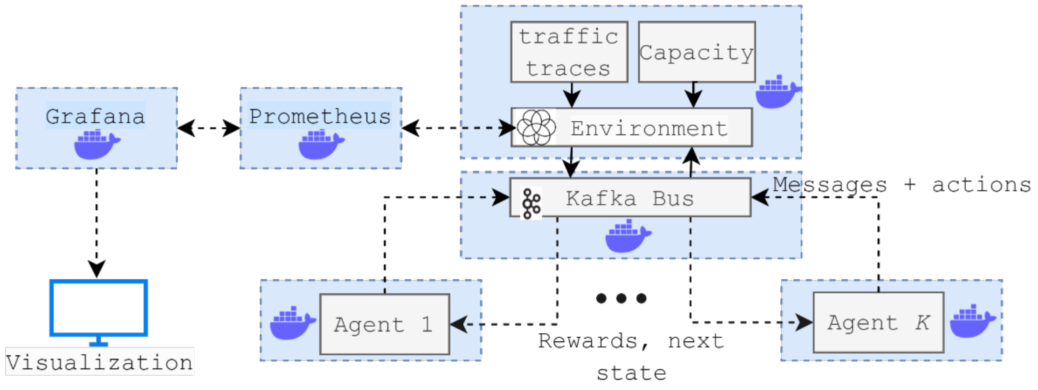


Figure 4: Cloud-native architecture for communication protocol learning

Cloud-native designs align well with the evolving needs of 5G and future 6G networks, which demand high flexibility, scalability, and resilience. By leveraging containerization, microservices, and continuous monitoring, the proposed architecture in Figure 4 can efficiently support the diverse and stringent requirements of different network slices, ranging from ultra-reliable low-latency communications (URLLC) to enhanced mobile broadband (eMBB) and massive machine-type communications (mMTC). This modular and scalable architecture is also future-proof, enabling seamless integration with emerging technologies and facilitating the transition towards fully autonomous and self-optimizing networks in the 6G era. The different blocks of the architecture are detailed in the sequel.

2.2.1. Agents Containers via Docker-Compose

A cloud-native architecture is introduced to manage network slices in future 6G environments [3], utilizing a collaborative multi-agent communication framework. Each network slice is controlled by an independent agent (client) that operates within a containerized environment, using Docker as the application abstraction layer. The containerization of these agents ensures that each operates in an isolated environment, providing a lightweight, scalable, and efficient mechanism to manage resources across multiple slices. Dockerization also facilitates the deployment, scaling, and orchestration of these agents across a distributed cloud infrastructure, making it easier to adapt to varying workloads and network demands inherent in 5G and 6G use cases.

2.2.2. Communication through Kafka Bus

The inter-agent as well as agent-server communication takes place over a Kafka Bus, which runs on a dedicated container. It allows the different agents and the server to produce and consume various types of events/data that are categorized by topics. This includes specifically the communication messages between agents to learn a protocol on-the-fly as well as the broadcast of rewards and next states by the environment server. Note that Kafka's scalability and asynchronous communication make it ideal for distributed systems [4]. It handles high data volumes by scaling horizontally, allowing services to produce and consume messages independently without direct synchronization. This decoupling enhances system resilience, reduces bottlenecks, and supports real-time data processing with high reliability and efficiency. The agents/Kafka configuration is given by the docker-compose file of Figure 5.

```
version: '3'
services:
  zookeeper:
    image: 'bitnami/zookeeper:latest'
    ports:
      - '2181:2181'
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
    logging:
      driver: "none"  # Suppressing logs completely
  kafka:
    image: 'bitnami/kafka:latest'
    ports:
      - '9092:9092'
    environment:
      - KAFKA_BROKER_ID=1
      - KAFKA_LISTENERS=PLAINTEXT://:9092
      - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
      - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT
      - ALLOW_PLAINTEXT_LISTENER=yes
    volumes:
      - ./scripts:/opt/bitnami/scripts/kafka/custom-scripts
    depends_on:
      - zookeeper
    command: bash -c "exec /opt/bitnami/scripts/kafka/run.sh > /dev/null 2>&1"
    logging:
      driver: "none"
  server:
    build: ./server
    depends_on:
      - kafka
  client1:
    build: ./client
    environment:
      CLIENT_ID: 1
    depends_on:
      - kafka
  client2:
    build: ./client
    environment:
      CLIENT_ID: 2
    depends_on:
      - kafka
  client3:
    build: ./client
    environment:
      CLIENT_ID: 3
    depends_on:
      - kafka
```

Figure 5: Docker-compose with agents and Kafka Bus configuration

2.2.3. Server Container

The agents communicate with a central server that functions as the information exchange hub. This centralized server ensures that while agents collaborate to optimize the infrastructure's overall performance, they do so without directly sharing sensitive information with each other. This design is crucial in maintaining privacy and security, as each agent remains unaware of the specific resource allocations and operational strategies of other agents. The central server aggregates and processes the data, ensuring that the collaborative decision-making process respects the constraints of the shared infrastructure and adheres to the privacy requirements of each network slice. The server dependencies are given in the Dockerfile of Figure 6.

```
# Use the official Python image from the Docker Hub
FROM python:3.8-slim
# Set the working directory in the container to /app
WORKDIR /app
# Copy the current directory contents into the container at /app
COPY . /app
# Install any needed packages specified in requirements.txt
RUN pip install kafka-python numpy pandas
# Run server.py when the container launches
CMD ["python", "server.py"]
```

Figure 6: Dockerfile for the server

2.2.4. Prometheus and Grafana

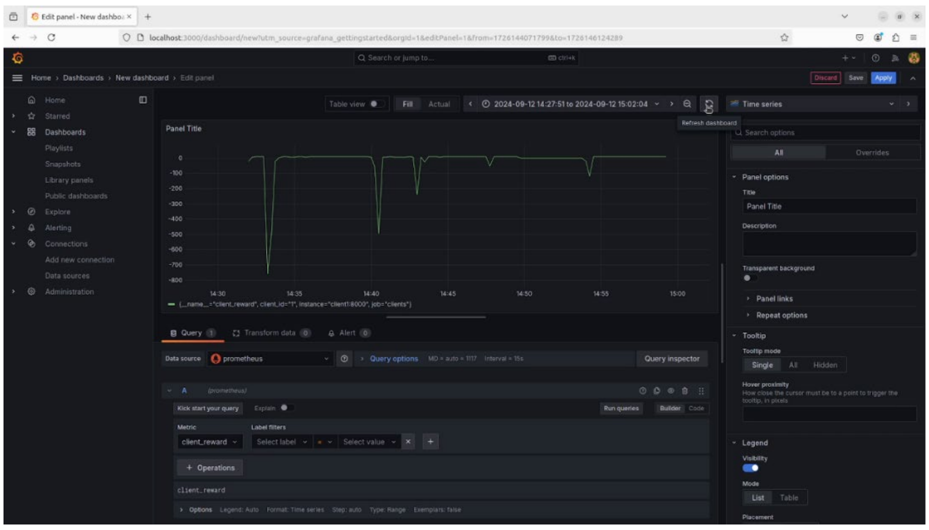
A key advantage of this cloud-native architecture is its ability to be monitored and managed in real-time using advanced monitoring and visualization tools like Prometheus and Grafana. Prometheus collects detailed metrics from each agent, allowing for real-time performance analysis, anomaly detection, and resource optimization. Grafana, in turn, provides a user-friendly interface for visualizing these metrics, offering insights into the operational status of the entire system. This online supervision capability is particularly valuable in dynamic 5G and 6G environments, where network conditions and demands can change rapidly, requiring swift and informed decision-making. An excerpt of Prometheus/Grafana configuration is given in Figure 7.

```
prometheus:
  image: prom/prometheus:latest
  volumes:
    - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
  ports:
    - '9090:9090'
  networks:
    - my_custom_network
grafana:
  image: grafana/grafana:latest
  ports:
    - '3000:3000'
  networks:
    - my_custom_network
```

Figure 7: Prometheus and Grafana configuration

2.2.5. Results of the Cloud-Native Architecture

Preliminary Grafana plots have been obtained for various metrics. Figure 8 for instance shows the increasing reward trend across episodes.



localhost:3000/dashboard/new?utm_source=grafana_gettingstarted&orgId=1&editPanel=1&from=1726144071799&to=1726146124289

2024-09-12 14:27:51 to 2024-09-12 15:02:04

{__name__="client_reward", client_id="1", instance="client1:8000", job="clients"}

Data source: prometheus
Query options MD = auto = 1117 Interval = 15s
A (prometheus)
Metric: client_reward
Label filters: Select label = Select value

```
client_reward
```

Options Legend: Auto Format: Time series Step: auto Type: Range Exemplars: false

Title: Panel Title
Description:
Transparent background
Tooltip mode: Single All Hidden
Visibility
Mode: List Table

Figure 8: Grafana plot of reward

2.3. Integration with the Slice Manager

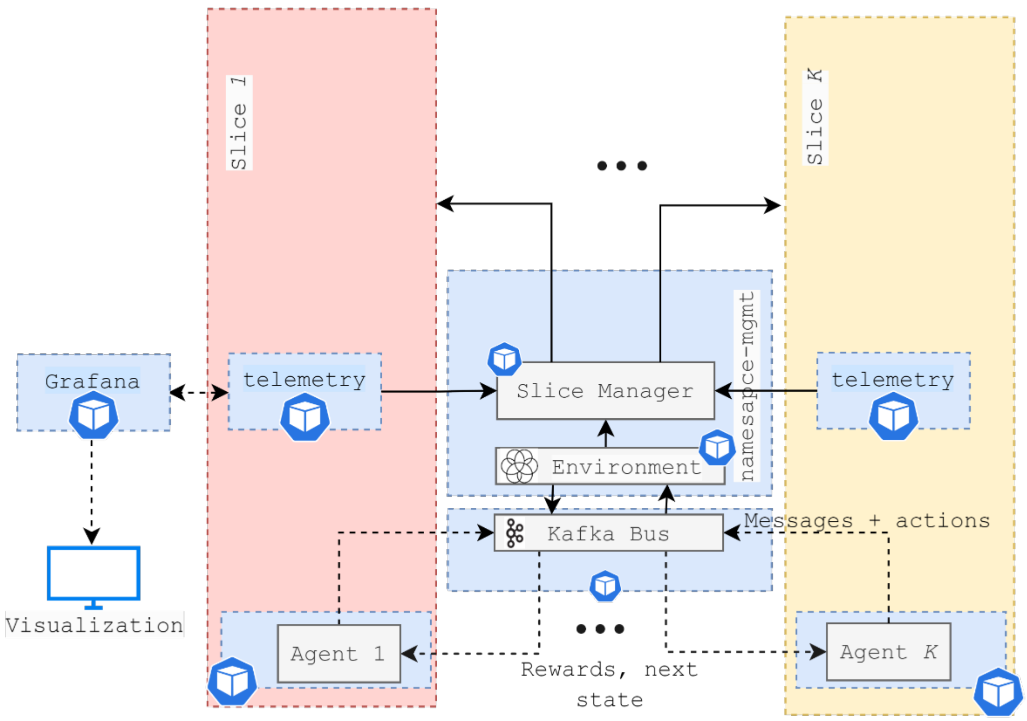
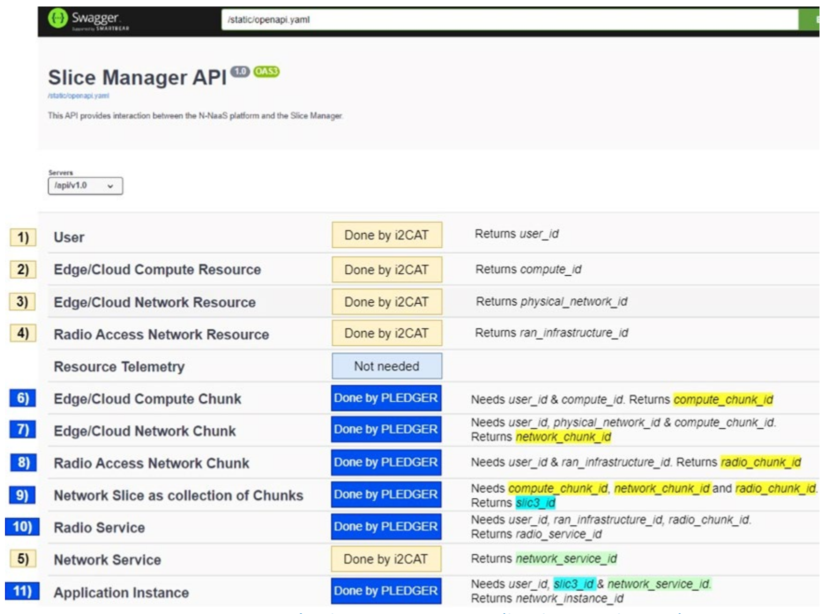


Figure 9: K8s-based integration of the AI Virtualiser with the Slice Manager

Through the Slice Manager (SM) rich API, the distributed AI Virtualiser consisting of the above-mentioned per-slice agents in Figure 9, will be able to enforce various slice reconfiguration and resource allocation actions to network slices. These slices are composed of computing, networking and radio chunks. The compute component is an isolated namespace.



Swagger /static/openapi.yaml

Slice Manager API 1.0 OAS3

/static/openapi.yaml

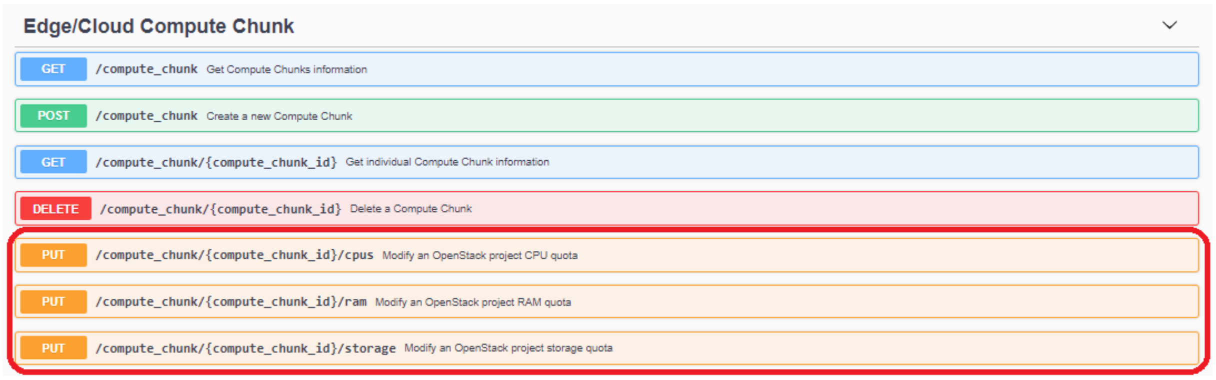
This API provides interaction between the N-NaaS platform and the Slice Manager.

Servers: /api/v1.0

1)	User	Done by i2CAT	Returns *user_id*
2)	Edge/Cloud Compute Resource	Done by i2CAT	Returns *compute_id*
3)	Edge/Cloud Network Resource	Done by i2CAT	Returns *physical_network_id*
4)	Radio Access Network Resource	Done by i2CAT	Returns *ran_infrastructure_id*
	Resource Telemetry	Not needed	
6)	Edge/Cloud Compute Chunk	Done by PLEDGER	Needs *user_id* & *compute_id*. Returns *compute_chunk_id*
7)	Edge/Cloud Network Chunk	Done by PLEDGER	Needs *user_id*, *physical_network_id* & *compute_chunk_id*. Returns *network_chunk_id*
8)	Radio Access Network Chunk	Done by PLEDGER	Needs *user_id* & *ran_infrastructure_id*. Returns *radio_chunk_id*
9)	Network Slice as collection of Chunks	Done by PLEDGER	Needs *compute_chunk_id*, *network_chunk_id* and *radio_chunk_id*. Returns *slic3_id*
10)	Radio Service	Done by PLEDGER	Needs *user_id*, *ran_infrastructure_id*, *radio_chunk_id*. Returns *radio_service_id*
5)	Network Service	Done by i2CAT	Returns *network_service_id*
11)	Application Instance	Done by PLEDGER	Needs *user_id*, *slic3_id* & *network_service_id*. Returns *network_instance_id*

Figure 10: i2CAT's Slice Manager API (background asset)

The environment in this case is the interface with the API gateway, translating various agents' actions into API calls to the Slice Manager (see Figures 10, 11 and 12). 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.



Edge/Cloud Compute Chunk

GET `/compute_chunk` Get Compute Chunks information

POST `/compute_chunk` Create a new Compute Chunk

GET `/compute_chunk/{compute_chunk_id}` Get individual Compute Chunk information

DELETE `/compute_chunk/{compute_chunk_id}` Delete a Compute Chunk

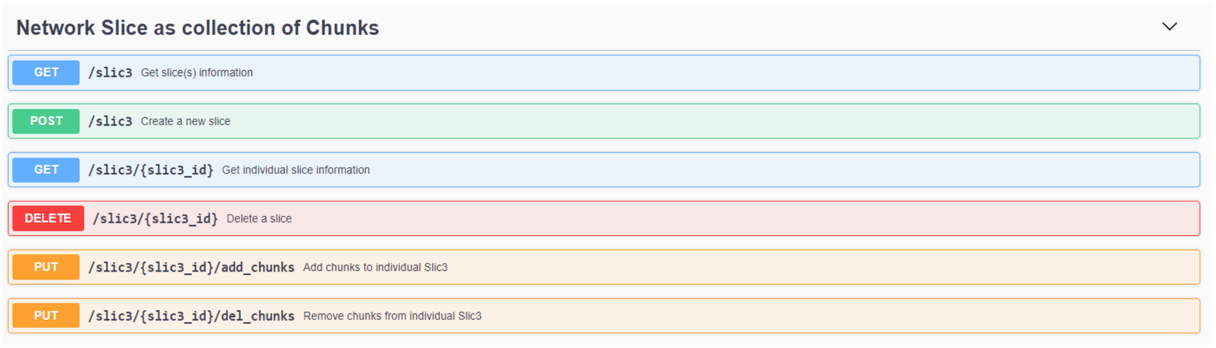
PUT `/compute_chunk/{compute_chunk_id}/cpus` Modify an OpenStack project CPU quota

PUT `/compute_chunk/{compute_chunk_id}/ram` Modify an OpenStack project RAM quota

PUT `/compute_chunk/{compute_chunk_id}/storage` Modify an OpenStack project storage quota

Figure 11: Edge/Cloud Chunk API

Created Slices can be edited by adding/removing chunks (referred by *chunk_id*), following a set of requirements for slice composition/update logic (for example, every instance must have at least one compute chunk; network chunk of type data-network is compulsory for slices with OpenStack 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).



Network Slice as collection of Chunks

GET `/slic3` Get slice(s) information

POST `/slic3` Create a new slice

GET `/slic3/{slic3_id}` Get individual slice information

DELETE `/slic3/{slic3_id}` Delete a slice

PUT `/slic3/{slic3_id}/add_chunks` Add chunks to individual Slic3

PUT `/slic3/{slic3_id}/del_chunks` Remove chunks from individual Slic3

Figure 12: Network Slice API

3. AI Virtualiser and Edge-level Resource Exploitation

The AI Virtualiser explores novel NFV-based virtualization techniques with the purpose of utilizing to the maximum extent the available resources at the network edge. In the new-generation 5G networks, the computational tasks are offloaded in the form of VNFs to powerful edge servers in close proximity to end-users, as the MEC model mandates [5]. Virtualizing the various network services according to the NFV standards has resulted in decoupling these services from specialized and proprietary hardware, which brings great flexibility and scalability in the overall network management [6] [7].

Aiming towards the maximum exploitation of the capabilities at the edge, the AI Virtualiser takes into consideration the current architectural nuances within the edge domain. Specifically, NANCY focuses on the peculiarities of the ARMv8 architecture which leads when it comes to edge deployments since it can inherently provide in an energy-efficient way, fast and parallel processing of computationally-intensive tasks, such as image processing, machine learning and cryptography [8]. Having this in mind, NANCY re-shapes the way virtualization is implemented when such devices are targeted. In these regards, NANCY designs a novel, lightweight virtualization solution for the edge of the network, which aims to guarantee greater isolation compared to existing alternatives. To provide some insights that will be further detailed in Section 3.2, with this virtualization solution NANCY achieves to deploy each VNF at an individual “partition” (subset) of an edge server’s hardware resources.

We are referring to such subsets of the edge server’s resources with both the terms “partitions” and “compartments”, interchangeably. In ARMv8, it is feasible to assign to specified “partitions” the execution of separate Operating Systems, which can execute simultaneously on-top of the same device [9]. In this context, at each “partition” NANCY deploys a VNF (which comes in the form of a Linux OS), that undergoes the execution of a computational task for implementing a 5G network service, like localization, anomaly detection, video streaming and others. Thanks to this setup, NANCY achieves improved isolation because the VNFs are deployed in individual “partitions” and do not affect the rest of the system. What is more, this virtualization solution is lightweight because it achieves near bare-metal execution times, since most parts of the physical hardware (e.g., memory, CPU) remain non-virtualized.

In the context of the AI Virtualiser, NANCY focuses on connecting the novel edge-based virtualization solution to the rest of the 5G infrastructure. In these regards, the AI Virtualiser introduces the *vManager*, which plays the role of the hypervisor that bridges a high-level 5G orchestrator with the low-level edge firmware that is privileged enough to instruct the actual deployment of the VNFs to the “partitions”.

In the upcoming sections, we will first describe the requirements needed for integrating a novel virtualization solution with a high-level 5G orchestrator such as OpenStack. Then, we will thoroughly detail the *vManager* hypervisor, along with its components and its internal APIs.

3.1. Integration with High-level Orchestrators

In the 5G ecosystem, NG-SDN and NFV have emerged as standard technologies that have permanently transformed traditional network services into virtualized entities [10]. Indeed, NG-SDN has enabled network services to be fully realized through softwarized components which, in line with the NFV terminology, consist of either individual VNFs, or chains of interconnected VNFs. In this environment, a resource orchestrator must be held responsible for determining the optimal placement of these software components within the infrastructure, by properly assigning the VNFs to the available computational resources [11].

Complying with the NFV architecture according to the ETSI MANO [12] standards (Figure 13), the resource orchestration is collaboratively achieved through parts of the NFVO, the VNFM and the VIM. Regarding the individual role of each component, the NFVO is at the top layer of the stack and is in charge of managing the life-cycle of the network services. The NFVO operates across multiple domains and undergoes global resource management, aiming to achieve the on-boarding, instantiation, scaling and termination of the network services. Then, the VNFM is responsible for the life-cycle management of the VNFs themselves. In particular, it manages the instantiation, scaling, updating and termination of the VNFs and it also carries-out KPIs monitoring upon them. Finally, in what concerns the VIM, this component operates in a network sub-domain and closely orchestrates the resources of the underlying NFVI. So, the VIM is in charge of controlling and managing the NFVI compute, network and storage resources, in the context of a single domain.

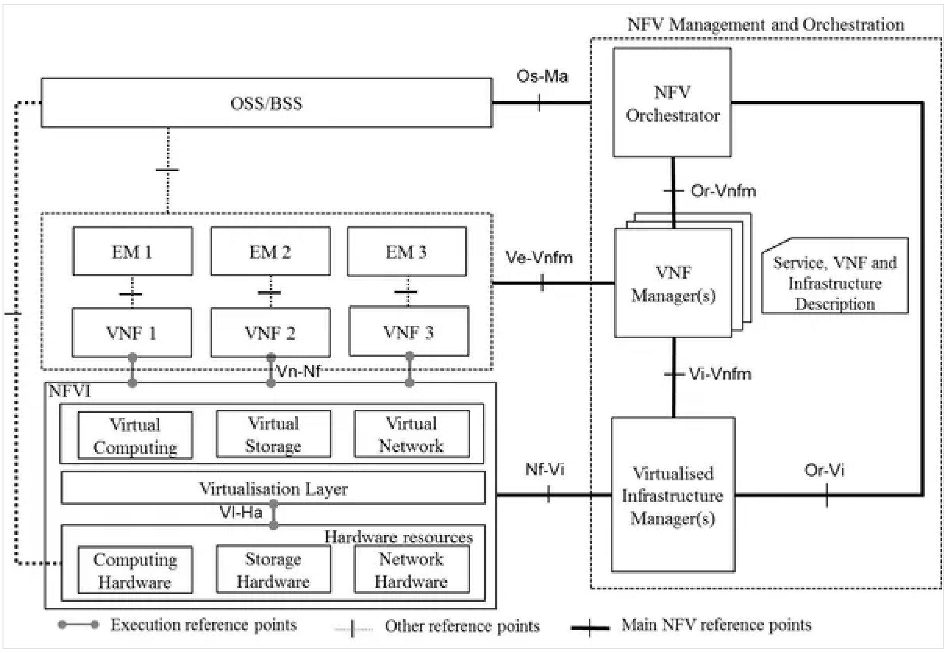


Figure 13: The ETSI-MANO NFV architecture

At the VIM layer, OpenStack is used as the de-facto commercial solution [13]. Leveraging a large number of open-source projects, OpenStack has emerged as the simplest solution for handling VIM deployments as it can provide full control over the storage, compute and networking resources. In what concerns the computational resources, the OpenStack Compute service (Nova) is a major component of the IaaS system [14]. Nova is the project that supports the management of the compute instances, which in reality are virtual servers. So, in its core, Nova provides all the low-level functionalities that are related to the virtual servers management, including their creation, destruction, scaling etc.

Underlying, Nova integrates with a variety of different hypervisors and thus it can support multiple virtualization solutions for the realization of the compute instances. The available hypervisors are implemented in Nova as Python drivers and the Nova deployment on the compute node is configured accordingly, to utilize a particular driver. Libvirt is the most commonly used hypervisor driver in Nova, which makes it the most commonly used virtualization driver for NFV deployments. Nova has included support for Libvirt that is backed-by KVM, Qemu, LXC and Virtuozzo. However, for official NFV deployments, Libvirt is mostly backed by Qemu, and in case it is available, KVM. The configuration to Nova compute node that instructs it to utilize Libvirt through, for example, Qemu should be appended to *"/etc/nova/nova.conf"* as follows (Figure 14):

```
# /etc/nova/nova.conf

[DEFAULT]

# COMPUTE

compute_driver=libvirt.LibvirtDriver

# LIBVIRT

[libvirt]

virt_type=qemu
```

Figure 14: Nova configuration for Libvirt with Qemu

When developing a novel virtualization solution aimed at NFV deployments in a network infrastructure, one should foresee that this solution should be finally embedded in the OpenStack Nova. Only by doing so, the VIM will be able to leverage this virtualization solution for the implementation of the virtualized entities. In the context of the AI Virtualiser, the novel virtualization solution suited for the Edge should similarly be able to integrate with Nova. In order to achieve this in an as-easy-as-possible way, we took the decision to integrate the new virtualization solution into Libvirt. This is also reasonable given that the deployment of the virtualized entities takes place through a Linux-based system on the embedded ARMv8 board, where the Libvirt daemon can execute unbothered. Then, following the directions of the already-supported Qemu/KVM, LXC and Virtuozzo Libvirt backends in Nova, the final integration should be a task that is trivial enough.

Thanks to the integration of the virtualization solution with the OpenStack Nova service, high-level resource orchestrators can enforce their decisions directly on the vManager "partitions", where the VNFs are deployed. In these regards, the decisions associated with compute resources that take place at the Slice Manager layer of the AI Virtualiser and which have been explained in Section 2 can be eventually enforced on the targeted nodes through the OpenStack Compute service (Nova). For example, a Slice Manager decision for removing a compute chunk from a Slice will translate into a Nova action that mandates the deletion of a vManager "partition" through Libvirt.

3.1.1. The Libvirt vManager Driver

The new hypervisor solution, which we call *vManager*, is integrated into Libvirt as a new hypervisor driver, following the Libvirt guidelines. We introduce the *virtvmand* service in Libvirt for this purpose, which encapsulates the functions of the *vManager* hypervisor driver.

The *virtvmand* service is accessible through the "***vman:///system***" URI, through which usual *virsh* commands targetting the *vManager* hypervisor are supported. The main objective of the vManager Libvirt driver is to finally communicate with lower layers of the vManager stack, using JSON objects. Libvirt has inherent support for JSON serialization / de-serialization functions, which allows hypervisor developers to create and issue commands of various types and wait for replies. Following is an example command "*create-partition*" serialized into a JSON object, issued and then waiting for a reply, from the lower layers of the vManager (Figure 15).

```
// Serialize the command
cmd = virVMANMonitorJSONMakeCommand("create-partition",
                                        "U:cores", maxvcpus,
                                        "U:primary", 0,
                                        "b:secure", false,
                                        "U:memsize", total_memory,
                                        NULL);

// Issue the command and wait for the reply
if (virVMANMonitorJSONCommand(mon, cmd, &reply) < 0)
   return -1;
// Check reply for errors
if (virVMANMonitorJSONCheckErrorFull(cmd, reply, true) < 0)
   return -1;
// Obtain the returned data
data = virJSONValueObjectGetObject(reply, "return");
// Parse the data
ret = virJSONValueObjectGetNumberUlong(data, "id", (unsigned long long *)&data_id);
```

Figure 15: Example create-partition command to demonstrate JSON functions utilization in Libvirt

The Libvirt vManager driver is registered on Libvirt as a set of callback functions wrapped in a *virHypervisorDriver* instance. These functions range from functions responsible for setting up the client connections through the URI, to informative functions regarding the status of connections and domains, down to the most meaningful functions that translate to specific instructions, in the form of JSON commands, towards the underlying layers of the vManager hypervisor.

For simplicity, Figure 16 only depicts the most basic callback functions that translate to final vManager hypervisor commands. The connection between these functions with the *virsh* commands and the internal vManager actions are further detailed in Section 3.3.2.

```
/* The vmanHypervisorDriver instance and basic callbacks that pass-through the vManager */
static virHypervisorDriver vmanHypervisorDriver = {
    .name = "VMAN",
    .domainCreate = virVMANDomainCreate,                   /* 7.5.0 */
    .domainShutdown = virVMANDomainShutdown,               /* 7.5.0 */
    .domainReboot = virVMANDomainReboot,                   /* 7.5.0 */
    .domainSuspend = virVMANDomainSuspend,                 /* 7.5.0 */
    .domainResume = virVMANDomainResume,                   /* 7.5.0 */
    .domainDestroy = virVMANDomainDestroy,                 /* 7.5.0 */
    .domainDefineXML = virVMANDomainDefineXML,             /* 7.5.0 */
};
```

Figure 16: Basic Libvirt virHypervisorDriver callbacks that interface with the vManager.

3.2. The vManager and its Internal API

The *vManager* is an ARMv8-compatible hypervisor solution that enables the dynamic definition and afterwards management of the ARMv8 system “partitions”. As already introduced, at these “partitions” or “compartments” which are subsets of the physical hardware resources, NANCY will deploy individual VNFs that implement computational network services. In these regards, the “partitions” are created with the purpose of hosting execution environments for various OSes in general, and VNFs in particular in the 5G stack.

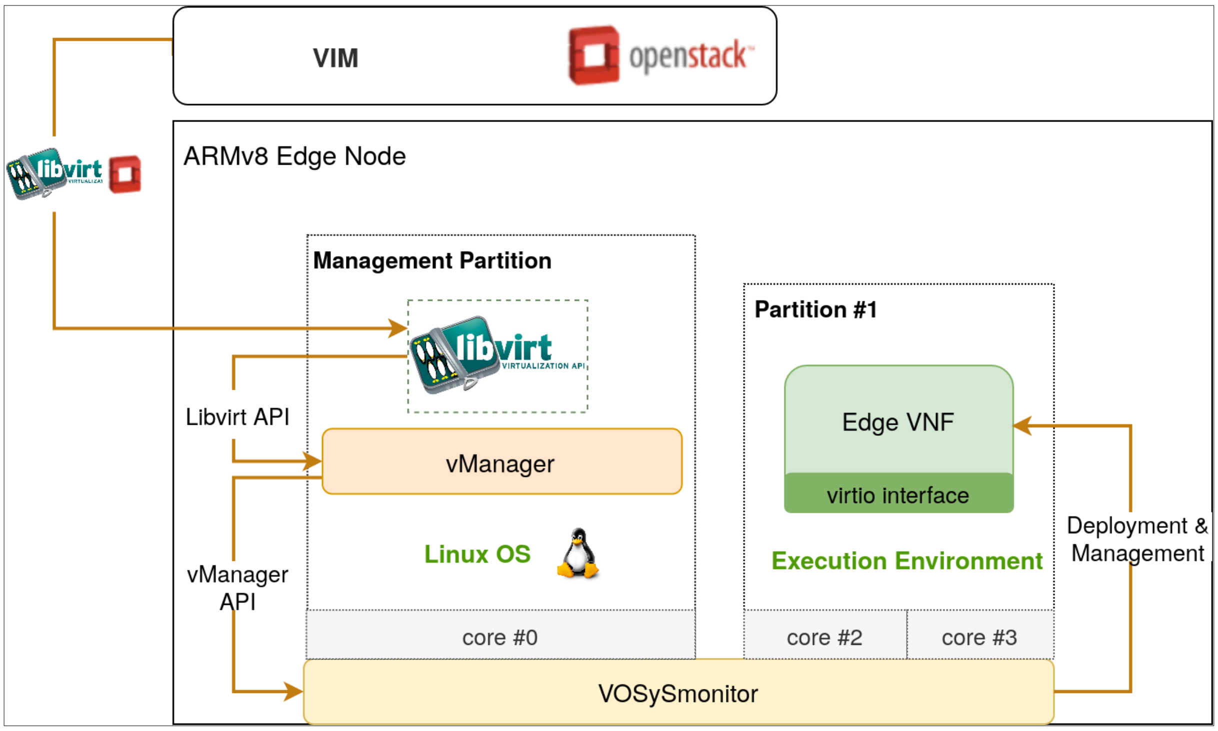


Figure 17: High-level depiction of the vManager hypervisor that undergoes the management of “partitions” to host individual VNFs.

Figure 17 depicts the overall architecture, from OpenStack to Libvirt to the *vManager* hypervisor that results in the management of “partitions” where VNFs are being deployed. As it is shown, *VOSySmonitor* is the software component that undergoes the actual deployment and management of the “partitions”. VOSySmonitor is a minimal low-level firmware which runs bare-metal in an ARMv8 board and has the privilege to define and control the “partitions” of the hardware resources by deploying OSes on them [9].

In the core of the ARMv8 architecture, there exist the ARM Trustzone hardware security extensions [15]. These extensions provide support for complete hardware-supported isolation of “critical tasks” (aka “secure”) from “non-critical tasks” (aka “non-secure”). The VOSySmonitor software is built upon these hardware extensions, with the initial aim of offering transparent and simultaneous execution of tasks with different critical levels. However, it should be noted that although the ARMv8 architecture only supports two levels of criticality (critical and non-critical), it is possible to deploy more than two “partitions” with the same critical level, on the same hardware platform.

To provide more technical details, VOSySmonitor runs at the highest “Execution Level” (EL3) of the ARMv8 architecture and is a “critical task”. This gives it the privilege to control and manage the resources of the system, as well as the allocation of the system resources to the various “partitions”. When the system boots, VOSySmonitor holds the responsibility to instantiate many of the system peripherals, given the use-case and the configuration it needs to apply for the “critical” and the “non-critical" worlds. It should also be noted that the "critical" partitions always have priority over the "non-critical" partitions, and VOSySmonitor guarantees that with its internal scheduling policies.

As also shown in Figure 17, the vManager hypervisor developed in the context of the AI Virtualiser establishes the connection point between Libvirt and the VOSySmonitor firmware. Given that VOSySmonitor is the highest-privileged (EL3) low-level bare-metal component directly deployed in the ARMv8 board, the vManager needs to implement a set of software components with increasing privilege levels (from user-space EL0 to kernel-space EL1) to finally interact with VOSySmonitor (at EL3). Specifically, the vManager consists of five separate components with different privileges that cooperate in order to achieve hardware partitioning based on user requirements:

1. The **virtvmand** at user-space (EL0), the Libvirt driver that interfaces with the vManager daemon through JSON messages (explained in Section 3.1).
2. The vManager daemon **vmand**. It is the user-space (EL0) background program that exchanges messages with *virtvmand* and forwards to the kernel-space the corresponding partition management commands. The available commands will be further described in Subsection 3.2.1).
3. The vManager controller **vmanctl**. It is the user-space (EL0) CLI tool that allows a user to execute directly the partition management commands described in Subsection 3.2.1 towards the kernel-space, by-passing entirely Libvirt.
4. The **vManager driver**. A loadable kernel module (kernel-space EL1) that enables the communication path between the VOSySmonitor firmware (EL3) and user-space applications (EL0). The vManager driver is the central component that implements the core logic of the vManager approach.
5. The **VOSySmonitor**. The firmware that executes on EL3, the highest privilege mode, and allows the co-execution of multiple Oses at different "partitions" of the same hardware platform.

The vManager daemon, controller and driver are deployed within a custom Linux distribution called *management partition*. The management partition is the first Operating System that is executed by VOSySmonitor during boot and is responsible for all the management operations with respect to other partitions.

The overall approach and the interconnection of various vManager sub-systems are depicted in Figure 18.

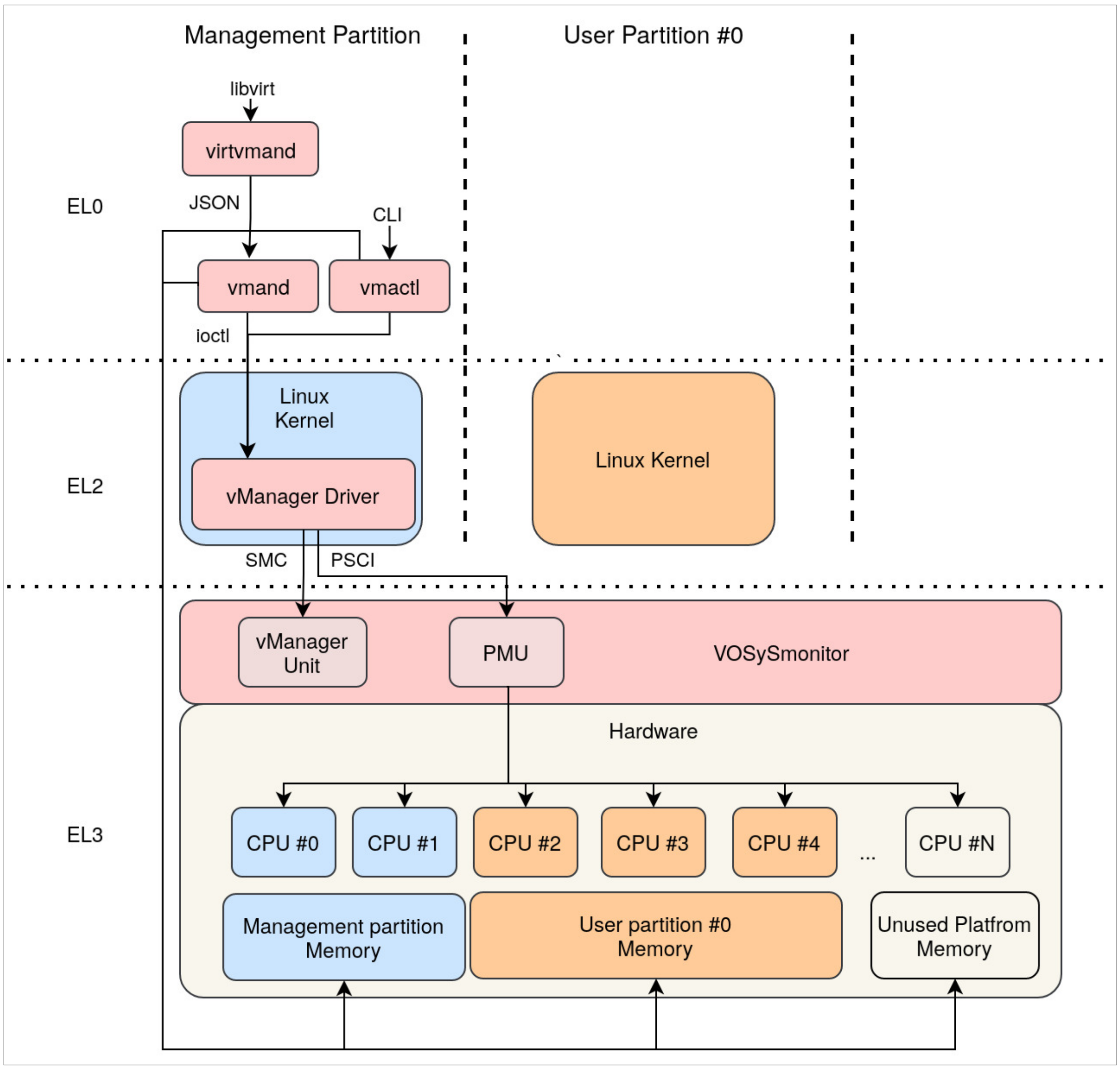


Figure 18: The vManager architecture

The *virtvmand* handles and parses messages from the Libvirt API, translating them into JSON messages that the vManager daemon can parse and execute as equivalent commands. The *vmand* daemon (or the *vmanctl* CLI application) communicates with the *vManager Driver* through ioctl commands on */dev/vmanager*. These ioctls finally reach the VOSySmonitor *vManager Unit* through the execution of the corresponding SMC, following the ARM SMC Calling Convention [16].

Following the creation of a partition, a new device appears under */dev/vmanX*, where X is the partition ID. This device activates additional ioctl commands for each partition. In the case that a power management command needs to be executed towards a partition (i.e., Shutdown or Reboot), a PSCI command is sent to VOSySmonitor that reaches the *Power Management Unit*.

3.2.1. The vManager Internal API

The internal API of the vManager comprises of all the necessary operations related to low-level management of the partitions.

3.2.1.1. Create a Partition

This is the command to create a partition. A user can specify how many cores as well as how much memory will be assigned to the partition that is being created. The user also defines whether the partition is “critical” (secure) or “non-critical” (non-secure). The vManager driver dynamically assigns specific CPUs, a memory space and an ID to the newly created partition.

The JSON API that is passed from *virtvmand* to the lower-layers for executing the command is:

```
“command”: “create-partition”,

“args” : {

    “cores”: <number_of_cores>,

    “secure”: <true/false>,“memsize”: <partition_memory_GB>

}
```

For a manual approach, from within the management partition, utilizing the *vmanctl* CLI, the user can execute:

```
vmactl  create-partition  —cores  <number_of_cores>  —memsize  <partition_memory_GB>  —secure
<true/false>
```

Following a successful creation the vManager marks the partition as **READY**.

3.2.1.2. Deploy a Partition

After a partition has been created, the user can instruct its execution by specifying the partition ID and providing the device tree path and the kernel to be utilized. The vManager loads the kernel and the device tree into the memory and instructs VOSySmonitor to bring up the CPUs. In case there is no device tree provided, the vManager driver will create one and load it into the partition’s memory.

The JSON API for executing the command is:

```
“command”: “deploy”,

“args” : {

    “kernel”: “path/to/kernel_image”,

    “dtb”: “path/to/device_tree_blob”,

    “partition_id”: <partition_ID>

}
```

Alternatively, from within the management partition, utilizing the *vmanctl* CLI, the user can execute:

```
vmactl deploy —kernel path/to/kernel_image —dtb path/to/device_tree_blob <partition_ID>
```

In order for the deployment to succeed, the selected partition state should be READY. Following a successful deployment, the partition state switches to **DEPLOYED**. A DEPLOYED partition cannot be re-deployed.

3.2.1.3. Destroy a Partition

An existing partition can be destroyed at any time, as long as there is no OS that is currently executing on-top. This command instructs VOSySmonitor to release the assigned hardware resources, which were previously occupied by the partition with the given ID.

The JSON API for executing the command is:

```
"command": "destroy-partition",

"args" : {

    "id": <partition_ID>

}
```

Alternatively, from within the management partition, utilizing the *vmanctl* CLI, the user can execute:

```
vmactl destroy-partition <partition_ID>
```

It should be noted that a DEPLOYED partition cannot be destroyed.

3.2.1.4. Reboot a Partition

A partition that is executing can be rebooted at any time utilising this command. This results in the forced termination of the execution of the CPUs that belong to this partition. Subsequently, the driver automatically reloads the deployment assets—the kernel and device tree—into the partition's memory, and the CPU is instructed to start the OS execution.

The JSON API for executing the command is:

```
"command": "reboot",

"args" : {

    "id": <partition_ID>

}
```

Alternatively, from within the management partition, utilizing the *vmanctl* CLI, the user can execute:

```
vmactl reboot <partition_ID>
```

This command can be only applied to a DEPLOYED partition.

3.2.1.5. Terminate a Partition

A partition that is executing can be terminated at any time utilising this command. This halts the CPU's execution within the partition without releasing any resources. The partition can then be redeployed with the same or a different OS.

The JSON API for executing the command is:

```
“command”: “shutdown”,

“args” : {

    “id”: <partition_ID>

}
```

Alternatively, from within the management partition, utilizing the *vmanctl* CLI, the user can execute:

```
vmactl shutdown <partition_ID>
```

This command can be only applied to a DEPLOYED partition.

3.2.1.6. Suspend a Partition

This command puts the CPUs of the specified partition into a low-power state, preserving their state for efficient resumption while reducing power consumption.

The JSON API for executing the command is:

```
“command”: “suspend”,

“args” : {

    “id”: <partition_ID>

}
```

Alternatively, from within the management partition, utilizing the *vmanctl* CLI, the user can execute:

```
vmactl suspend <partition_ID>
```

This command can be only applied to a DEPLOYED partition. Following a successful suspension, the partition state switches to **SUSPENDED.**

3.2.1.7. Restore a Partition

This command brings the CPUs of the specified partition out of the SUSPENDED state, restoring their previous state and resuming normal execution.

The JSON API for executing the command is:

```
“command”: “restore”,
"args" : {
    "id": <partition_ID>
}
```

Alternatively, from within the management partition, utilizing the *vmanctl* CLI, the user can execute:

```
vmactl restore <partition_ID>
```

This command can be only applied to a SUSPENDED partition. Following a successful restoration, the partition state switches to **DEPLOYED.**

The overall state transition diagram for the states of the partitions is depicted in Figure 19.

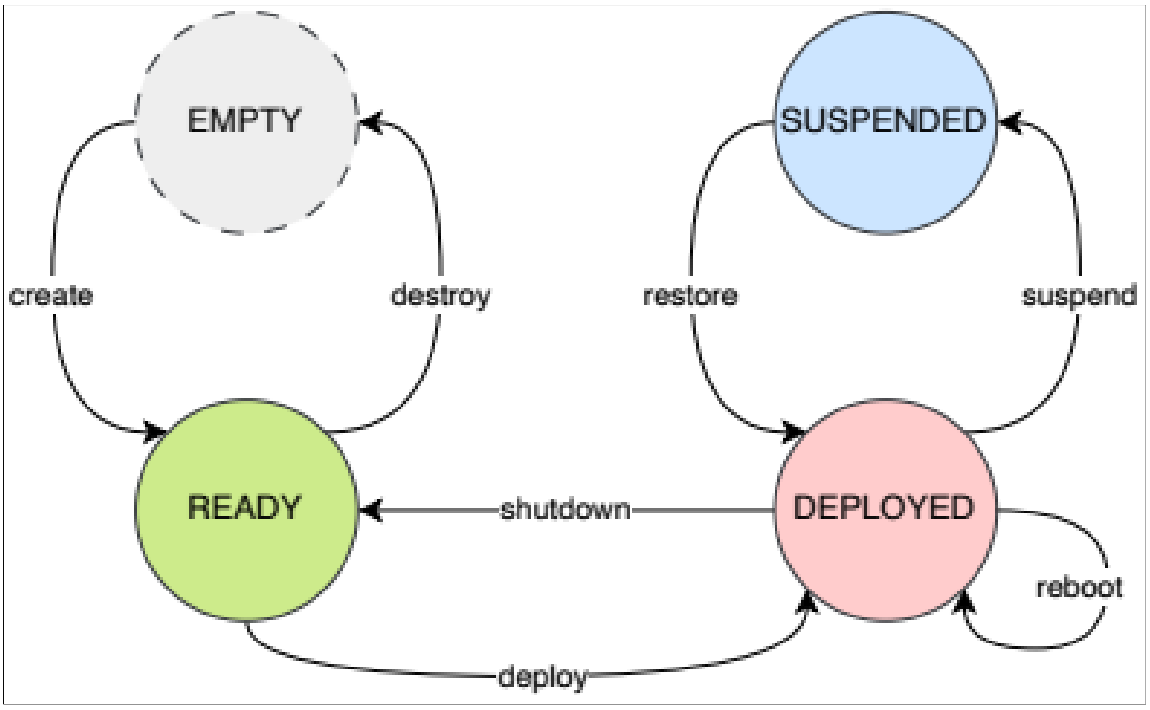


Figure 19: The state transition diagram depicting the possible partition states.

3.2.2. Virsh Commands and the vManager API

Libvirt exposes the low-level hypervisor functionalities to third-party applications, such as Nova, through a set of daemons. As already mentioned in Section 3.2.1, any third-party application can interact with the vManager hypervisor after establishing a connection through the "***vman:///system***" URI. Then, by utilizing the *virsh* tool, it is possible to directly interface with the vManager components.

The association between the *virsh* commands and the vManager equivalent functionalities is depicted in Table 1.

Table 1: Mapping between virsh commands and the vManager API

virsh command	virtvmand driver function	vManager equivalent
virsh define	virVMANDomainDefine	create-partition
virsh start	virVMANDomainCreate	deploy
virsh create	virVMANDomainCreate	create-partition & deploy
virsh shutdown	virVMANDomainShutdown	shutdown
virsh destroy	virVMANDomainDestroy	destroy
virsh reboot	virVMANDomainReboot	reboot
virsh suspend	virVMANDomainSuspend	suspend
virsh resume	virVMANDomainResume	restore

4. AI Virtualiser and Inter-domain CPU Resource Provisioning

This chapter introduces the solutions provided by NANCY for managing the performance of software workloads in Linux. These software workloads include activities such as VNFs that are, for example, encapsulated in a container of software processes executing in KVM/QEMU virtual machines.
The context of NANCY often requires different computational activities to share the same computing resources (e.g., computing nodes and processing cores) while: **(1)** providing QoS guarantees on their real-time performance **(2)** not causing the underutilization of the computing nodes.
In NANCY, this is achieved by means of the reservation-based scheduling provided by the SCHED_DEADLINE scheduling class of Linux, which is briefly recalled in the following.

4.1. Underutilisation of Virtualized Computational Resources on Linux

SCHED_DEADLINE. SCHED_DEADLINE [17] is a Linux scheduler that implements a resource reservation mechanism to allow encapsulating applications into virtual platforms, each with a guaranteed fraction of the core CPU bandwidth and with a bounded CPU service delay [18] [19]. In essence, SCHED_DEADLINE guarantees CPU resource isolation and a configurable CPU bandwidth with bounded service delay (latency) for general Linux tasks.

The resource reservation mechanism implemented by SCHED_DEADLINE is based on the Constant Bandwidth Server algorithm [20] and both provide a resource partitioning and an enforcement mechanism. The latter feature is particularly important in the context of NANCY, where different applications may not trust each other. SCHED_DEADLINE ensures that each application receives no more than the allocated CPU bandwidth, thus shielding other applications from possible misbehaviours of malicious (or simply bugged) applications that may otherwise harm their real-time behaviour.

SCHED_DEADLINE works by assigning to each managed entity two parameters: a period P and a budget Q (also called runtime). SCHED_DEADLINE then ensures that no more than Q time units of processing time are provided to the application every period P. The applications managed by SCHED_DEADLINE can be either Linux threads and processes, virtual machines (e.g., KVM/QEMU), or containers. The last two options are shown in Figure 20. Threads, processes and VMs are supported by mainline Linux: instead, an out-of-tree kernel patch (contributed by SSS) is required to manage containers.

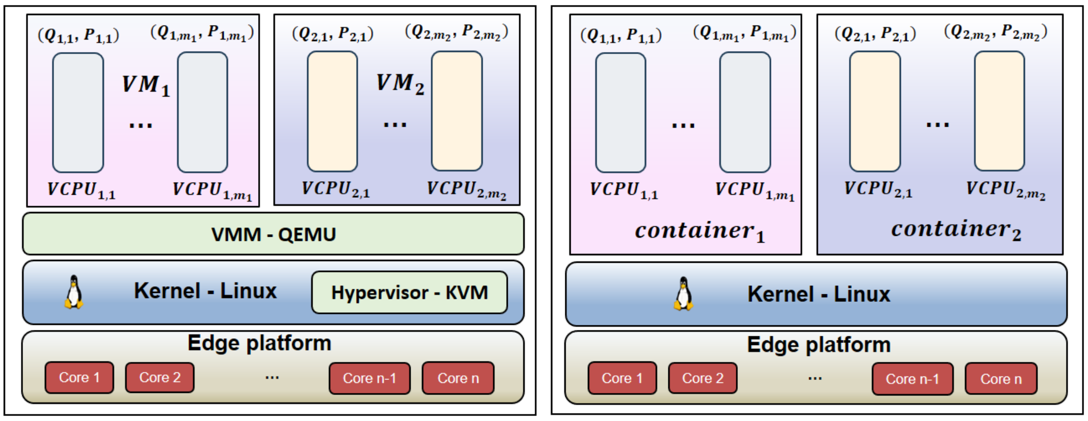


Figure 20: Different usages of SCHED_DEADLINE reservations for virtualized workloads.

Each SCHED_DEADLINE reservation server must be properly configured with the two key parameters Q and P. The theoretical properties of SCHED_DEADLINE make a link between these values and the fraction of CPU bandwidth and the CPU service delay provided to the workload running in the reservation. In particular:

- $\alpha = \frac{Q}{P}$ is the CPU bandwidth provided to the reservation;
- $\Delta = 2 * (P - Q)$ is the CPU service delay.

When reservations are properly configured, SCHED_DEADLINE allows different applications to be co-located in the same platform (e.g., edge node) thanks to the resource isolation mechanism (implemented by budget enforcement). Figure 21 shows the advantage: two (temporally)-untrusted applications requiring both three cores in parallel to perform a parallel operation, one with a requirement of 50% of the core bandwidth and one with 20% would be allocated to six cores using classical coarse-grained allocation solutions (i.e., each application is exclusively allocated to an appropriate number of dedicated cores to avoid CPU interference). Differently, SCHED_DEADLINE allows co-locating multiple applications on three cores, also leaving some spare bandwidth for other tasks.

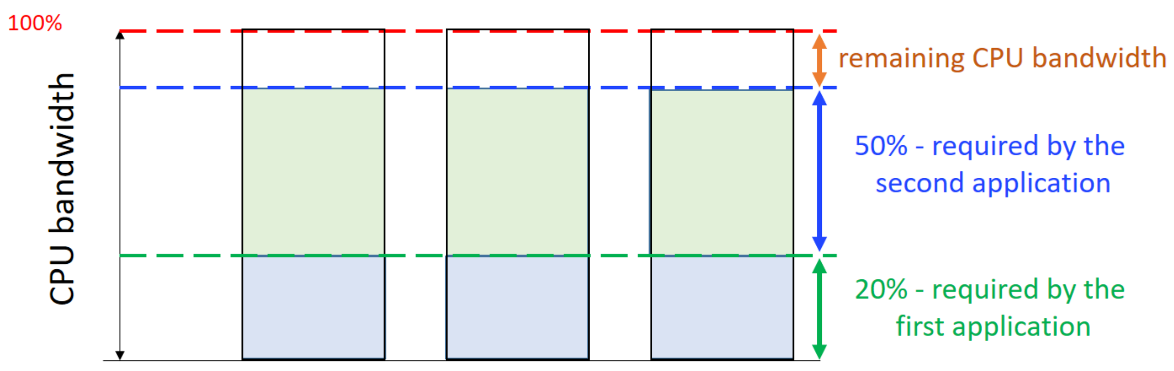

100%
CPU bandwidth
remaining CPU bandwidth
50% - required by the second application
20% - required by the first application

Figure 21: Usage of SCHED_DEADLINE to co-locate multiple applications on the same cores while providing timing isolation.

Clearly, SCHED_DEADLINE needs to be properly configured with the aforementioned budget and period parameters to provide the desired properties. Indeed, an inaccurate configuration of a reservation may lead to the following issues:

- If the budget is too small or the period is too large real-time constraints cannot be guaranteed.
- If the budget is too large or the period is too small the edge node can be underutilized.

In traditional (safety-critical) real-time systems, the Q and P parameters are set based on the temporal properties of the tasks running within the reservation, i.e., the (maximum) execution time and the period, which are known a-priori. However, the situation is much more complex in the context of NANCY, in which the workloads are dynamic. Indeed, platform-specific execution time estimates are often unknown or not accurate. Applications can even be not periodic. Hence, tools for estimating these parameters are developed. The problem of configuring the period of a SCHED_DEADLINE reservation is addressed in Task 4.2.

This deliverable focuses on the configuration of the budget parameter, providing an integrated solution to monitor the computational requirements of the virtualized workload running inside the reservation and tuning the budget accordingly. The provided solution directly uses the mechanisms provided by SCHED_DEADLINE and is suitable for dynamic workloads (such as those of NANCY) because it does not require code instrumentation.

4.2. Runtime Monitoring of vCPUs

When scheduling one or more activities (single threads, processes, containers, VMs, ...) with SCHED_DEADLINE (possibly using the real-time control group scheduling patchset [21]), the experienced QoS can be controlled if the tasks' periods and runtimes are known. As previously discussed, the QoS of virtualized software workload is controlled by a budget and period parameter. A tool for determining the suitable period is derived in Task 4.2. Nevertheless, even with a suitable period, the problem of monitoring the tasks' runtimes still needed to be addressed. This information is crucial to set the budget for SCHED_DEADLINE reservations.

This scheduling approach (dynamic detection of runtimes) is similar to feedback scheduling [18] (and adaptive reservations [22] in particular). However, adaptive reservations are implemented by reading some *observed value* (the so-called *scheduling error*) to be used by a control algorithm to set some actuator (the reserved runtime). Observing the scheduling error requires instrumenting the code to mark the beginning (and the end) of each real-time job. For example, a periodic task can be implemented as in Figure 22, where the wait_for_next_activation() call marks the end of each periodic activity.

```
while (!finished) {
    wait_for_next_activation();

    /* … do something… */
    /*     (job body)    */
}
```

Figure 22: The code structure of a classical periodic task.

Unfortunately, often it is not possible to know a-priori the structure of applications offloaded to edge nodes in NANCY, which could not follow this code structure. Hence, a different approach based on monitoring the execution time of each task has been adopted.

The monitoring mechanism is designed to be applied to generic code, even to non-real-time applications, and takes advantage of the SCHED_DEADLINE features. This scheduling policy allows reserving a “dl_runtime” (the budget) every “dl_period” (the period) to the scheduled task, which is guaranteed to be able to execute for such a CPU time, but is not allowed to execute for more. Then, the CPU time accounting mechanism implemented by the Linux kernel can be used to measure how much time the task actually executed for (a kernel patch is needed to export this information to user-space); if such time is smaller than “dl_runtime”, then the reserved runtime can be decreased, otherwise, it must be increased. The monitor is hence based on the following ideas:

- SCHED_DEADLINE is used to ensure that a task (or a group of tasks) can execute for at most Q time units (budget) every P (period).
- A kernel patch is used to periodically monitor the amount of execution time used by each task.
- *If a task (or group of tasks) executes for the maximum reserved time*, it is assumed that the reserved time is too small. Hence, Q is increased.

- If the initial value of Q is too small, then the tasks can accumulate some delay until the reserved time is enough:
  - To compensate this effect, if the tasks consume the whole reserved time for multiple times in a row, then the speed at which Q is increased is accelerated.
  - This feature is implemented by using a variable "l" which is doubled every time that executed time increases and is set to 1 when the executed time does not increase.

Notice that in order for the feedback algorithm to work, the allocated runtime cannot be set exactly to the maximum measured execution time, but it must be set to a slightly larger value (otherwise the tasks would not be able to execute for more than the detected time, and the allocated runtime would not increase). Hence, if some tasks execute for *C* time units over *P* time units, *Q* is not set as "*Q = C*" but as "*Q = C * (1 + ovh)*" (we use the name overallocation overhead for "*ovh*"). Finally, to account for the "l" term mentioned above, the equation is updated as:

$$Q = C * (1 + ovh * l)$$

The algorithm pseudo-code is reported in Figure 23.
In the algorithm, "N" is the size of the circular array (i.e., the number of previous samples used to compute the maximum) and "ovh" is the overallocation overhead (these two values are configurable parameters of the algorithm).

Implementation. This algorithm has been implemented in a runtime monitor which is able to monitor single threads (as described above) or groups of threads (using the Linux cgroups feature). Docker/Podman (or Kubernetes) containers are handled through their cgroups, while KVM-based VMs can be monitored by monitoring their virtual CPU threads (running a monitor as a daemon inside the VM is also possible). The monitor needs to be both efficient (introducing a small overhead) and safe (avoiding bugs due to wrong memory accesses), and the Rust language looked like a good compromise between efficiency and safety.
Hence, the monitor has been implemented in Rust.

Usage. The monitoring program can be used in different ways:
- As a wrapper that is able to start the monitored programs and monitor/manage them
- As a standalone program that can monitor one or more existing threads (also setting the reserved runtime)
- As a daemon providing a REST API, which can receive the IDs of the threads to be monitored

```
void MonitorAndUpdate(int P, int Q, int id) {

/* INPUTS: P (task period), Q (initial runtime estimation), id (task ID) */
/* The initial estimation Q does not need to be accurate:
   it will be adjusted by the algorithm */

   1. Schedule task "id" with SCHED_DEADLINE using parameters
      dl_runtime=Q and dl_period=P;

   2. Initialize l = 1; cmax_old = -1; t_old = -1;
      Initialize circular_array = circular array of size N;

   3. Start periodic monitoring with period T. Every T time units, do:

   t = get_current_time();
```

```
    c = get_executed_time(id);
    circular_array.insert(c);
    cmax = circular_array.get_max();

    if (cmax_old != -1) {
        if (cmax > cmax_old) {
            l = l * 2;
        }
    }
}
```

Figure 23: Pseudo-code of the monitoring approach.

Figure 24 shows the evolution of the runtime allocated to two periodic threads with execution time 10ms and 20ms (the two tasks have a period of 100ms and the monitoring period is 500ms). To stress the robustness of the algorithm, the original runtime estimations have been set to two completely underestimated values (2ms). As it is possible to see from the figure, the allocated runtime rapidly increases to a maximum value (set to 80% of the period), thanks to the multiplicative effect of the "l" variable; this allows recovering from the delay accumulated by the tasks in the initial periods when the allocated runtime was not large enough. After this delay is recovered, the allocated runtime decreases to a value slightly larger than the thread's execution time (this small overallocation is due to the "ovh" factor).

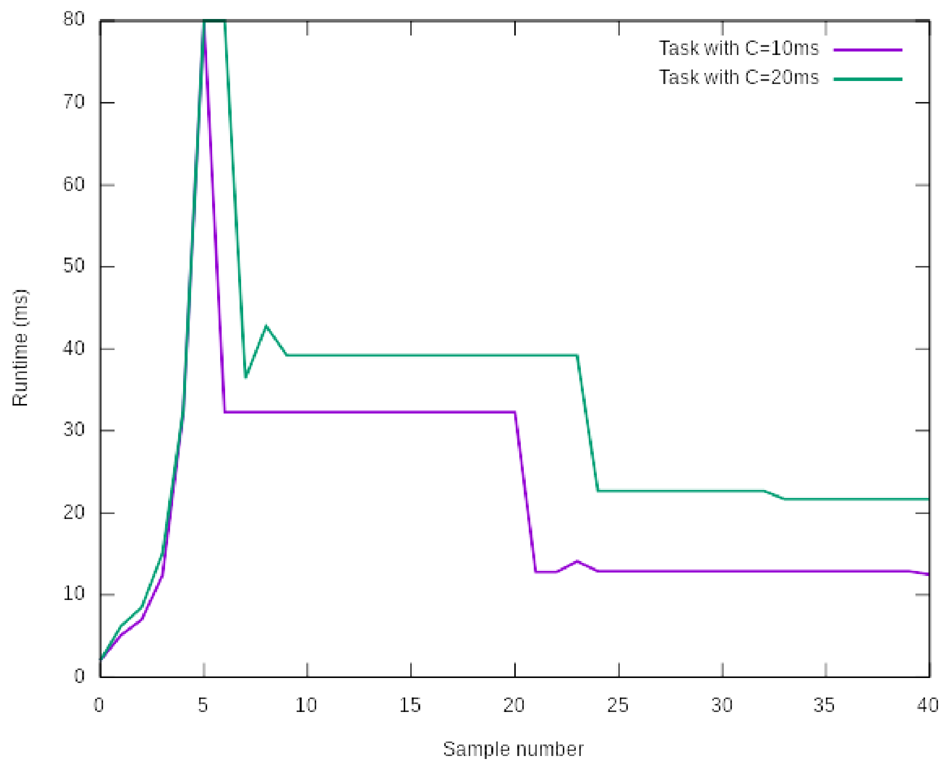


Figure 24: Runtime (budget) of the SCHED_DEADLINE reservations of two threads (initial runtime estimation = 2ms).

Figure 25 shows the evolution of the response times for the two tasks:

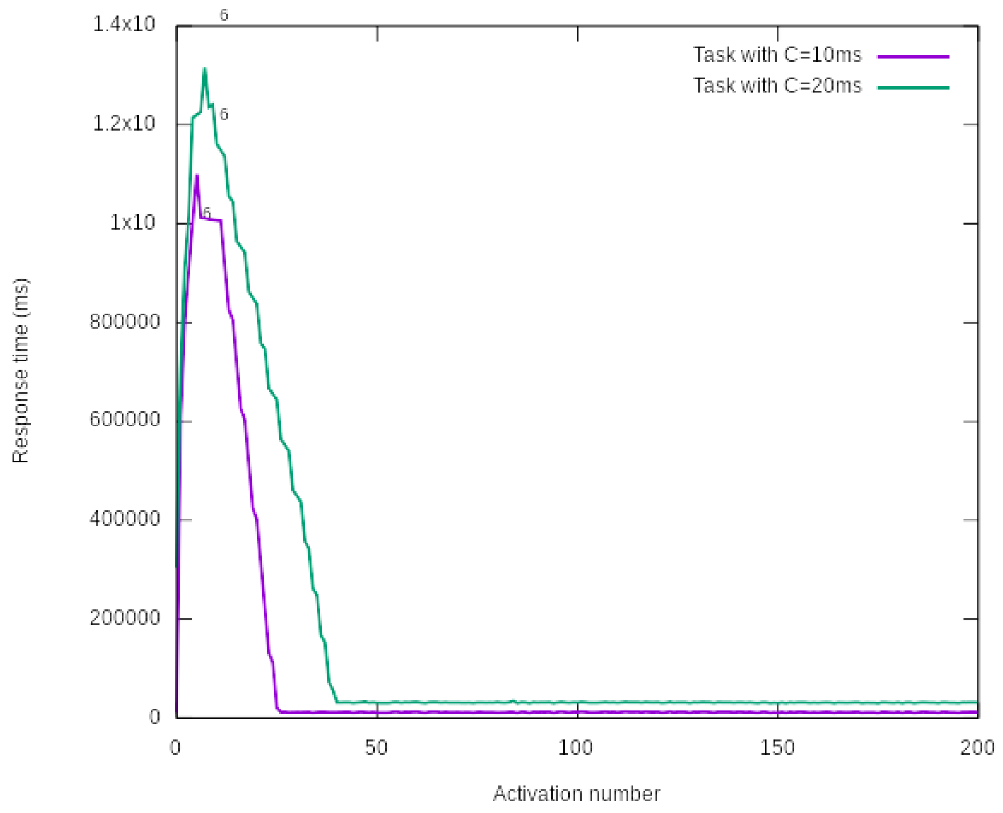


Figure 25: Response time of two threads within SCHED_DEADLINE (initial runtime estimation = 2ms).

Notice how the response times initially increase to very large values, but after some time (after a few monitoring cycles) they become equal to the threads' execution times, showing that the monitoring and feedback algorithms worked correctly.

Figures 26 and 27 show the results of a new run of the experiment in which the initial runtime estimation was more accurate (Q=8ms).

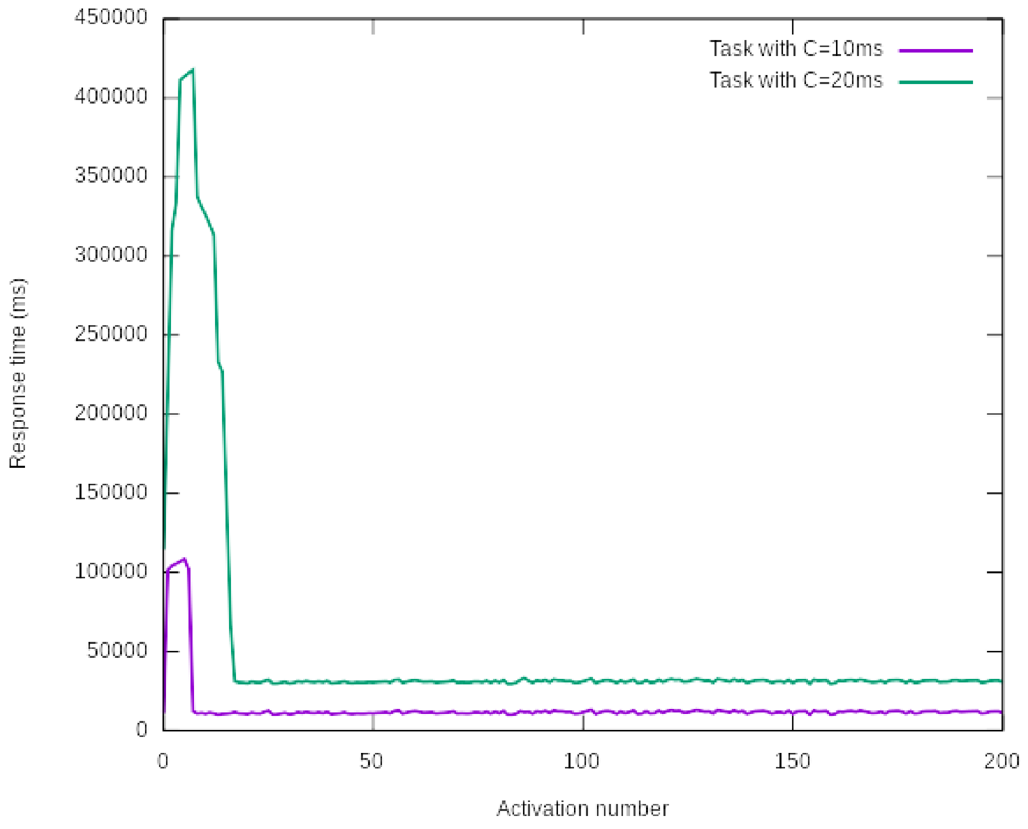


Figure 26: Runtime (budget) of the SCHED_DEADLINE reservations of two threads (initial runtime estimation = 8ms).

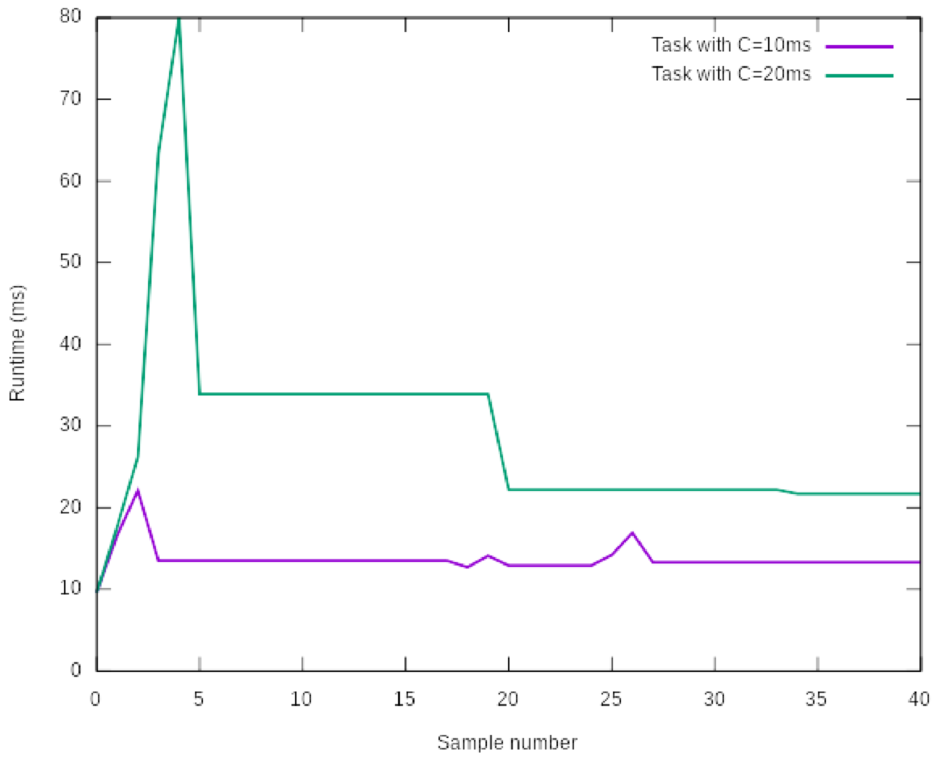


Figure 27: Response time of two threads within SCHED_DEADLINE reservations (initial runtime estimation = 8ms).

5. AI Virtualiser on the NANCY Functional and Deployment View

In this section, the main components of the AI Virtualiser will be placed into the NANCY overall architecture (depicted in Figure 28). Furthermore, in Section 5.1 the relation between the AI Virtualiser and the offloading decisions will be detailed.

On the slices management side, after training the AI Virtualiser agents at the level of the CI/CD Platform as it will be described in Section 7, they are deployed on the namespaces of the slices they control and connected to the Slice Manager (Resource Orchestrator) via a secured VPN to enforce the slice-level resource allocation via its exposed API as shown in Figure 9.

At the edge layer, the vManager solution of the AI Virtualiser, which reshapes virtualization for ARM devices by proposing a very lightweight implementation with strong isolation characteristics to better support the execution of virtualized functions, is mapped into the functional and deployment architecture under the NFVO. The Compute Controller in this case is the OpenStack Nova, as detailed in Section 3.2, which through Libvirt will enforce decisions coming from the orchestrators on the ARM targets.

On the other hand, SCHED_DEADLINE-based management to reduce underutilization of compute resources, as described in Section 4.1, is part of the Resource Orchestrator, which as specified in D6.1 is responsible for handling compute resources. The runtime monitoring of vCPU described in Section 4.2 is part of the Resource Orchestrator as well, as it is used to drive the orchestration of compute resources using SCHED_DEADLINE as an actuation means.

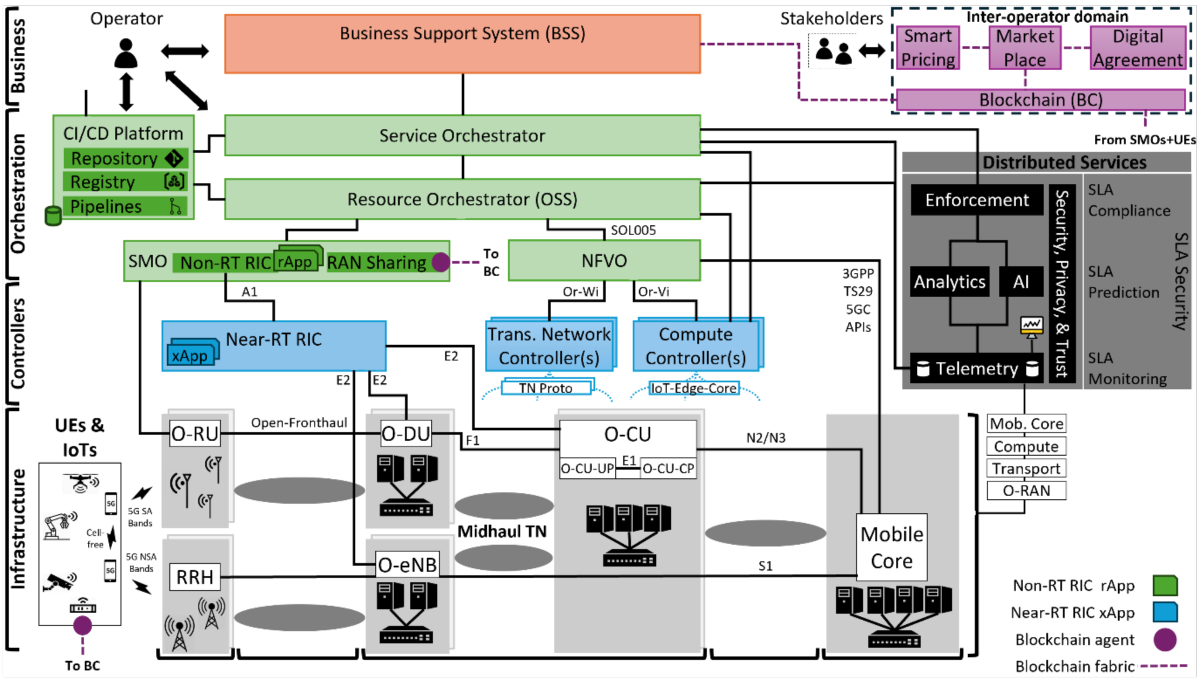


Figure 28: Functional and deployment view of the NANCY architecture.

5.1. AI Virtualiser and Offloading Decisions

Another key functionality enabled in NANCY architecture is task offloading, with a focus on selecting the best VNF placement and migration for continuous service provisioning, improving quality of service (QoS) and user experience (QoE). By enabling the dynamic placement of processing functions, and supported by data caching functions, task offloading mechanisms pave the way for flexible services that meet strict service level agreements (SLAs), in terms of latency, bandwidth, connectivity, data security, power consumption or reliability, in complex scenarios. This is achieved by an elastic computational resource allocation scheme that decides the best point in the end-to-end network infrastructure to run a given task. Besides, this decision is not permanent as the offloaded functions may be migrated to another point in runtime if performance benefits are predicted. This is particularly relevant for non-static scenarios, such as mobility services in vehicular environments as those addressed in NANCY's demonstrators and in-lab testbeds.

Figure 28 presents the Functional and deployment view of the NANCY architecture, which serves as the reference basis for enhanced coordination and implementation of every task, allowing to design of the interactions required between components to behave autonomously and intelligently under variable conditions. In this sense, the provisioning of task offloading capabilities is also built upon this architecture. Task offloading is a set of subsequent steps that focus on determining and deploying in the best location a given computational or networking load. To this aim, task offloading mechanisms consider the whole range of segments in a Beyond 5G infrastructure (far edge-edge-cloud). Furthermore, offloading mechanisms are designed to encompass both intra and inter-operator domains. Collaboration between operators through these mechanisms makes it possible to leverage under-utilised resources for use by an operator in need. Therefore, deployment of networking or processing functions can happen at the most suitable node, perhaps in an operator different from the one currently used by the user. Besides, aligned with offloading decision schemes, user-centric caching mechanisms will support task offloading by enhancing the data transfer performance, such as the seamless migration of virtualised services between different nodes through VNF-caching. As mentioned above, this is especially relevant in mobile scenarios where a sluggish resource migration can lead to the degradation or violation of an SLA in force. Considering that the SLA concept is pivotal in NANCY, the purpose of these processes is to meet the requirements set in the SLAs, thereby ensuring that contracts are fulfilled, and the user/costumer receives the expected level of quality when consuming the service.

Considering the NANCY architectural view presented in Figure 28, we briefly present the main components driving the offloading mechanisms. For a more detailed explanation, please refer to deliverable D4.1 "Computational Offloading and User-centric Caching".

Offloading is considered a particular case of orchestration in which solving an SLA associated with a service requires locating part of the service in specific nodes, moving them from the original centralized or constrained destination to the optimal one. Therefore, for task offloading to occur, an SLA evaluation needs to take place beforehand. During the SLA enforcement process, the KPIs are analysed to grant the proper runtime of the service. The AI/analytics engines decide how to continuously guarantee the requirements. Constant monitoring modules extract metrics and analytics to oversee the SLA, detecting potential risks. Among the risks addressed by offloading and caching mechanisms, we highlight the following, resource exhaustion at the node, decreased performance due to network congestion or user mobility. Envisioned analytics are capable of taking into consideration the needs behind the SLA fulfilment and composing an enforcement plan, placing the VNFs comprising the service chain in their optimal places. Migration mechanisms are task-specific task offloading mechanisms focused on moving an offloaded task between edge servers. Migration uses caching mechanisms to streamline the connection handover, avoiding interruptions in the service delivery. At any moment, if the provider operator compromises the SLA preservation, the marketplace is accessed to find other providers capable of maintaining the service requirements. The Smart pricing module propels the profitability of services and resources with intelligent and fair automated bidding system. The Digital Agreement Creator, supported by the blockchain, stores the result of the agreements providing trustworthiness to service consumption and accountability. The Service orchestrator takes the outputs generated after the AI-based decisions, deploying and configuring services to compose E2E network services.

Therefore, we can conclude that the task offloading management process involves a complex procedure that focuses on meeting users' needs while ensuring infrastructure stability. It analyses each offloading request to determine if the demanded task can be handled effectively, taking into account the resources available in the different domains involved. The decision engines select the most appropriate solution to meet the user's demands, considering aspects such as their location and needed resources. As a final step, the network orchestrator deploys and configures the offloaded task in the most efficient way to meet the user's requirements and keeps monitoring it to ensure its performance. As it can be understood, this complex process requires advanced decision-making mechanisms to support the orchestration functions. In general, with the constant and fast evolution of telecommunications, the use of intelligent orchestrators has become essential to optimize network performance and consequently improve the user experience. These orchestrators, powered by artificial intelligence and machine learning mechanisms, also enhance the efficiency and adaptability of the network to meet the user's and service's changing needs.

In this line, NANCY proposes the use of an AI virtualiser module, a building block that acts as a central node to support the overall intelligent orchestration system. It aims to improve the functionality of the orchestrator by integrating machine learning interactions, allowing it to effectively and efficiently identify computational requirements and orchestrate the allocation of both radio and computational resources, which is the foundation for decision-making in the task offloading workflow, as explained previously. By using innovative technologies and advanced ML algorithms, the AI virtualiser improves the agility of orchestrations and contributes to the overall process efficiency within the NANCY ecosystem. By ensuring that SLAs are met, the AI virtualiser helps to improve the quality of service provided by the offloaded tasks.

6. AI Virtualiser in NANCY Testbeds and Demonstrators

The components developed in the context of the AI Virtualiser will be tested in NANCY's testbeds and demonstrators, focusing on different use-cases and usage scenarios. Detailed descriptions of these setups are provided as follows.

6.1. Integration with the Spanish Demonstrator

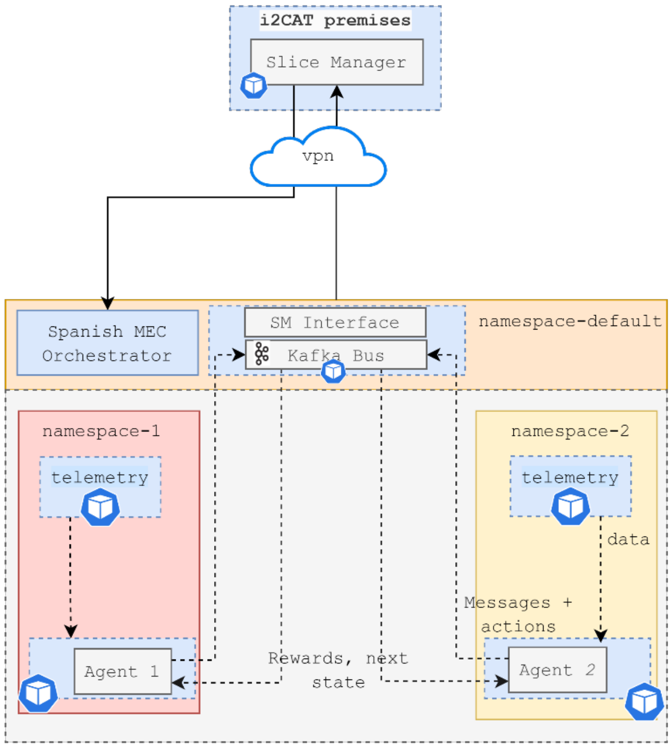


Figure 29: AI Virtualiser integration with the Spanish edge K8s cluster and MEC orchestrator

The Spanish demonstrator's testbed is composed of NFV and MEC infrastructures supporting various technologies and deployment options. Specifically, Kubernetes (K8s) clusters can be deployed on top of the Edge server orchestrated by a MEC orchestrator. As shown in Figure 29, the distributed AI Virtualiser consists of a set of pods: a telemetry server and an agent deployed on top of each slice (i.e., namespaces) to perform CPU resource management. The communication between these agents is ensured by a Kafka bus deployed on the default namespace, where an interface with the Slice Manager based on API calls is also set up. The Slice Manager is hosted on i2CAT premises and connected through VPN to the K8s cluster using the `kubeconfig` file. It can send commands to the MEC orchestrator which actuates on the infrastructure to perform various resource allocation tasks in specific namespaces.

Note that the generic slices 1 and 2 will correspond to services (such as URLLC and eMBB) targeted by the Spanish demonstrator.

6.2. Integration with the Italian Testbed

The Italian in-lab testbed will provide the environment to validate the novel NFV-based virtualization solution that is tailored to the characteristics of ARM devices at the network edge, in the context of the AI Virtualiser. For this purpose, the Italian in-labb testbed integrates an SK-AM69x Texas Instruments board as an ARMv8 edge server (depicted in Figure 30). The demonstration will specifically validate the vManager solution, testing its ability to provide individual bare-metal partitions where to host the Virtualized Network Functions (VNFs) in an isolated way. The VOSySmonitor firmware will be employed on the board as the low-level partitioner of the ARMv8 system resources, together with the higher-level vManager components, detailed in Section 3.2. The scenario will focus on demonstrating both the increased isolation that is granted by the bare-metal compartments compared to usual VM deployments, as well as the low execution latency of the solution.

In detail, the increased isolation will be showcased by employing in the ARMv8 board a Linux application provided by SSS that is able to simulate Denial-of-Service (DoS) attacks in the system. With respect to these attacks, the functioning of non-directly attacked partitions won't be compromised, as it would be the case with usual, non-hardware-isolated VMs. The overall performance of the solution will be measured in the context of the examined Video Streaming use-case for the Italian in-lab testbed. In particular, the use-case will be focused on offloading video streams to edge servers with available resources. In these regards, the low-execution latency of the novel edge virtualization solution will be showcased through the offloading of the ITL Video streaming application in the created vManager partitions and the collection of performance metrics accordingly.

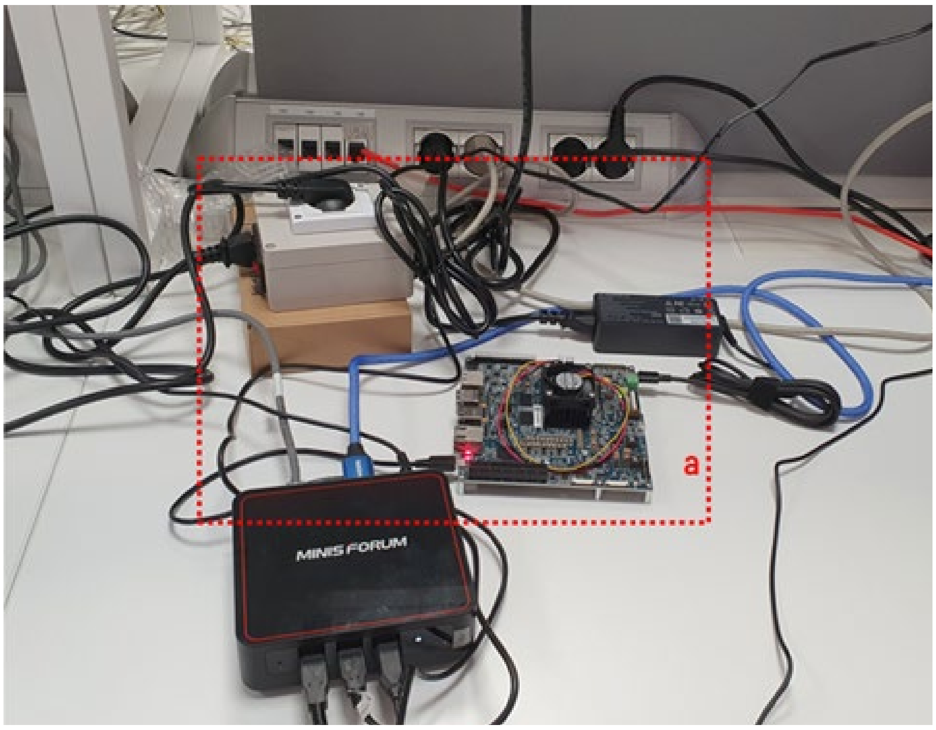


Figure 30: The SK-AM69x Texas Instrument board in the Italian in-Lab testbed.

7. CI/CD Integration of the AI Virtualiser and Testing

Continuous Integration and Continuous Delivery (CI/CD), encompasses a collection of best practices and methodologies in software development. The primary goal is to enhance the reliability of code changes while reducing development cycles through extensive use of automation.

Continuous Integration (CI) refers to developers merging their code changes regularly into a central repository [23]. Each update triggers automated builds and various tests through a CI server [24] to ensure application stability. In the context of NANCY, this process includes packaging software into Docker container images, storing them in a registry [25], and deploying them in a dedicated development/testing environment, specifically the central NANCY Kubernetes cluster. The primary goal is to speed up the release cycle by identifying and fixing bugs early, thus reducing extensive rework, allowing teams to focus more on development and integration.

Continuous Delivery (CD) is the subsequent phase after Continuous Integration in the software release process. It involves manually triggering the deployment to production environments through the CI server, once all CI workflows have been successfully validated. This stage is dedicated to preparing the software artifact for distribution to end-users in the NANCY testbeds and demonstration environments.

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.

In the context of NANCY AI Virtualiser, the CI/CD system is envisioned to be used for training and testing of the containerized Agent component described in Section 2.2 within the central NANCY development/testing environment, as well as its continuous delivery towards the NANCY testbed/demonstrator – specific deployment environments (i.e., Kubernetes clusters).

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 Agent container images respectively, as shown in Figure 31 and Figure 32.

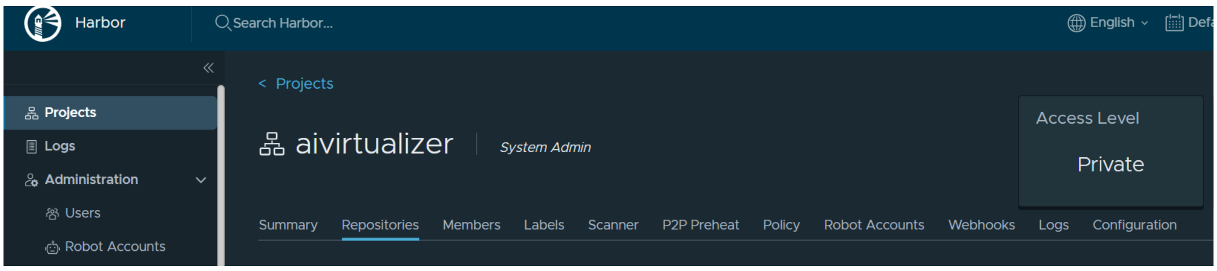
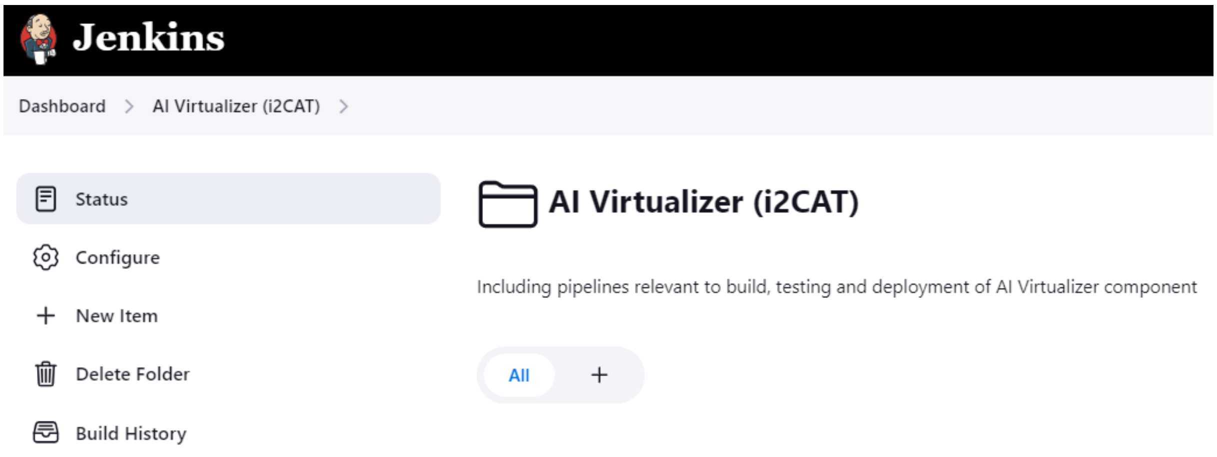


Figure 31: Dedicated Harbor project to host and manage AIVirtualizer container images



AI Virtualizer (i2CAT)

Including pipelines relevant to build, testing and deployment of AI Virtualizer component

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

8. Conclusion

This deliverable presents the development of the AI Virtualiser, a central component of the NANCY architecture that equips the orchestrator module with cutting-edge technologies and intelligent ML algorithms. The incorporation of the presented innovative solutions, powered by a suite of modern 5G technologies such as Slicing, NG-SDN and NFV, shows that the resource utilization is optimized in the networked infrastructure, by achieving the minimization of resources under-utilization while ensuring the QoS at multiple layers.

The AI Virtualiser is a multi-layered component that extends from the Slice Manager layer to fine-grained edge-centered innovations. At the Slice Manager layer, the presented innovations, backed by intelligent DRL techniques, achieve strategic and coordinated CPU allocation among different slices, demonstrating significantly improved CPU utilisation, elimination of resource conflicts between slices as well as increased latency performance. At the network edge, an innovative virtualization solution that focuses on ARM devices has been implemented and adjusted for utilisation through OpenStack in the NFV stack, achieving the maximum exploitation of the virtualized computational resources at this layer. Also focusing on the edge, a fine-grained CPU allocation mechanism has been presented, which efficiently controls the QoS of virtualized software workloads by informatively adjusting the CPU budget parameter while scheduling them at an edge node.

For the aforementioned technologies, implementation details have been thoroughly presented, accompanied by explicit code snippets and experimental results. Also, the validation of the AI Virtualiser as part of the NANCY testbeds and demonstrators has been described. Finally, the CI/CD testing of the Slice Manager innovations of the AI Virtualiser through the setup of dedicated workspaces has been detailed.

Bibliography

[1] M. Hervás-Gutiérrez, E. Baena, C. Baena, J. Villegas, R. Barco, and S. Fortes, "Impact of CPU Resource Allocation on vRAN Performance in O-Cloud," *TechRxiv,* 2023.

[2] F. Rezazadeh, H. Chergui, S. Siddiqui,J. Mangues, H. Song, W. Saad, and M. Bennis, "Intelligible Protocol Learning for Resource Allocation in 6G O-RAN Slicing," in IEEE Wireless Communications, vol. 31, no. 5, pp. 192-199, Oct. 2024.

[3] J. S. Camargo et al. "Toward Cloud-Native Protocol Learning for Conflict-Free 6G: A Case Study on Inter-Slice Resource Allocation," in *to be submitted to IEEE EuCNC 2025*.

[4] "How to Scale and Balance a Kafka Cluster," [Online]. Available: https://developer.confluent.io/courses/architecture/cluster-elasticity/

[5] A. Filali, A. Abouaomar, S. Cherkaoui, A. Kobbane, and M. Guizani, "Multi-access edge computing: A survey," *IEEE Access,* vol. 8, pp. 197017-197046, October 2020.

[6] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine,* vol. 53, pp. 90-97, February 2015.

[7] R. Mijumbi, J. Serrat, J.L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials,* vol. 18, pp. 236-262, September 2015.

[8] E. Krishnasamy, S. Varrette, and M. Mucciardi, "Edge computing: An overview of framework and applications," 2020.

[9] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, and D. Raho, "Vosysmonitor, a low latency monitor layer for mixed-criticality systems on ARMv8-A," in *ECRTS 2017*, 2017.

[10] I. Cerrato, A. Palesandro, F. Risso, M. Suñé, V. Vercellone, and H. Woesner, "Toward dynamic virtualized network services in telecom operator networks," *Computer Networks,* vol. 92, pp. 380-395, December 2015.

[11] J.G. Herrera and J.F. Botero, "Resource allocation in NFV: A comprehensive survey," *IEEE Transactions on Network and Service Management,* vol. 13, pp. 518-532, August 2016.

[12] ETSI NFV ISG, "Network Function Virtualization (NFV) Management and Orchestration," 2021.

[13] D. Haja, M. Szabo, M. Szalay, A. Nagy, A. Kern, L. Toka, and B. Sonkoly, "How to orchestrate a distributed OpenStack," in *IEEE INFOCOM 2018*, Honolulu, HI, USA, 2018.

[14] T. Rosado and J. Bernardino, "An overview of OpenStack architecture," in *IDEAS '14: 18th International Database Engineering & Applications Symposium*, Porto, Portugal, 2014.

[15] S. Pinto and N. Santosl, "Demystifying ARM TrustZone: A comprehensive survey," *ACM Computing Surveys (CSUR),* vol. 51, pp. 1-36, January 2019.

[16] ARM Limited, "Secure Monitor Call (SMC) Calling Convention," ARM Limited, October 2024. [Online].

[17] J. Lelli, C. Scordino, L. Abeni and D. Faggioli, "Deadline scheduling in the Linux kernel," vol. 46, no. 6, 2015.

[18] J. Stankovic, C. Lu, S. H. Son and G. Tao, "The case for feedback control real-time scheduling," in *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*, York, UK, 1999.

[19] I. Shin and I. Lee, "Compositional Real-Time Scheduling," in *IEEE International Real-Time Systems Symposium*, Lisbon, Portugal, 2004.

[20] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," in *Proceedings 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[21] L. Abeni, A. Balsini and T. Cucinotta, "Container-based real-time scheduling in the Linux kernel," *SIGBED Reviews,* vol. 16, no. 3, October 2019.

[22] L. Abeni, L. Palopoli, G. Lipari and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *IEEE Real-Time Systems Symposium*, Austin, TX, USA, 2002.

[23] "NANCY Github page," [Online]. Available: https://github.com/NANCY-PROJECT.

[24] "NANCY Jenkins server," [Online]. Available: https://jenkins.nancy.rid-intrasoft.eu.

[25] "Harbor containers registry," [Online]. Available: https://harbor.nancy.rid-intrasoft.eu/