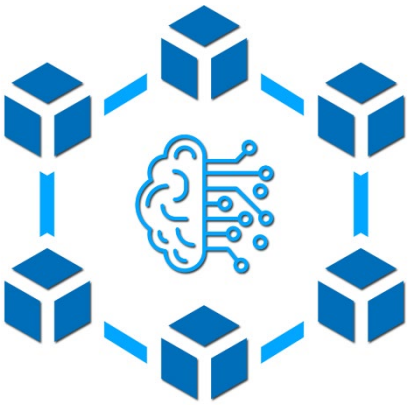


NANCY

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

Deliverable 4.2

Resource Elasticity Techniques

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	Resource Elasiticity Techniques
Deliverable Number	D4.2
Work Package	WP4
Associated Task	T4.2 Resource Elasticity & Adapticity Enabling Techniques
Dissemination Level	Public
Due Date	30 November 2024
Completion Date	29 November 2024
Submission Date	30 November 2024
Deliverable Lead Partner	JSI
Deliverable Author(s)	Emanuele De Santis (CRAT), Andrea Wrona (CRAT), Antonio Pietrabiassa (CRAT), Miguel Catalan Cid (i2CAT), Hatim Chergui (i2CAT), Carolina Fortuna (JSI), Shih-Kai Chou (JSI), Jernej Hribar (JSI), Jovan Prodanov (JSI), Maria Belesioti (OTE), Daniel Casini (SSS), Dimitris Manolopoulos (UBITECH), Panagiotis Sarigiannidis (UOWM), Thomas Lagkas (UOWM), Dimitrios Pliatsios (UOWM), Athanasios Liatifis (UOWM), Sotirios Tegos (UOWM)
Version	1.0

Document History

Version	Date	Change History	Author(s)	Organisation
0.1	31/5/2024	Initial Table of contents	Shih-Kai Chou, Carolina Fortuna	JSI
0.2	13/9/2024	Initial version	Shih-Kai Chou	JSI
0.3	24/9/2024	Adding to Section 4.3	Jovan Prodanov, Jernej Hribar, Shih-Kai Chou	JSI
0.3	2/10/2024	Adding to Section 2.2	Emanuele De Santis, Andrea Wrona, Antonio Pietrabiassa	CRAT
0.4	3/10/2024	Adding to Section 3.2	Dimitris Manolopoulos	UBITECH
0.5	7/10/2024	Adding to Section 4.2	Miguel Catalan Cid	i2CAT
0.6	8/10/2024	Adding to Section 2.1	Maria Belesioti	OTE
0.7	11/10/2024	Adding to Section 4.1	Daniel Casini	SSS
0.8	21/10/2024	Adding to Section 3.1	Hatim Chergui	i2CAT
0.9	5/11/2024	Adding exclusive summary, introduction, and conclusion	Shih-Kai Chou	JSI
0.9	7/11/2024	End-to-end checks, polish and formatting.	Jovan Prodanov, Carolina Fortuna, Shih-Kai Chou	JSI
1.0	28/11/2024	Addressing the comments from internal reviewers and quality check revisions	Jovan Prodanov, Carolina Fortuna, Shih-Kai Chou/ Emanuele De Santis/ Miguel Catalan Cid/ Daniel Casini/ Dimitris Manolopoulos/ Panagiotis Sarigiannidis, Thomas	JSI/ CRAT/ i2CAT/ SSS/

			Lagkas, Dimitrios Pliatsios, Athanasios Liatifis, Sotirios Tegos	UBITECH / UOWM

Internal Review History

Name	Organisation	Date
Maria Belesioti	OTE	14 November 2024
Anna Panagopoulou, Alvise Rigo	VOS	13 November 2024

Quality Manager Revision

Name	Organisation	Date
Anna Triantafyllou, Dimitrios Pliatsios	UOWM	29 November 2024

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Explanation
3GPP	3rd Generation Partnership Project
AI	Artificial Intelligence
API	Application Programming Interface
ARFCN	Absolute Radio Frequency Channel Number
B-RAN	Blockchain Radio Access Network
BPMN	Business Process Model and Notation
CAV	Connected Autonomous Vehicles
CG-MARL	Centralized-Global Multi-Agent Reinforcement Learning
CPU	Central Processing Unit
CU	Central Unit
DMDDPG	Deterministic Multi-Agent Deep Deterministic Policy Gradient
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
DU	Distributed Unit
eMBB	Enhanced Mobile Broadband
EDF	Earliest Deadline First
EE	Energy Efficiency
FaaS	Function as a Service
HetNet	Heterogeneous Network
IAB	Integrated Access and Backhaul
IIoT	Industrial Internet of Things
IoT	Internet of Things
KPI	Key Performance Indicator
K8s-aaS	Kubernetes as a Service
LaaS	Localization as a Service
LCM	Life Cycle Management
MADRL	Multi-Agent Deep Reinforcement Learning
MAFRL	Multi-Agent Federated Reinforcement Learning
MARL	Multi-Agent Reinforcement Learning
MEC	Multi-access Edge Computing
MFG	Mean Field Game
MIoT	Massive Internet of Things
MIMO	Multiple Input Multiple Output
ML	Machine Learning
MDP	Markov Decision Process
MT	Mobile Terminal
OSM	Open Source MANO (Management and Orchestration)
OSS	Operations Support System
PaaS	Platform as a Service
PPO	Proximal Policy Optimization

PRB	Physical Resource Block
QoE	Quality of Experience
QoS	Quality of Service
RAN	Radio Access Network
RIC	RAN Intelligent Controller
RL	Reinforcement Learning
SLA	Service Level Agreement
SRA	Spectrum Resource Allocation
TMF	TeleManagement Forum
UE	User Equipment
UAV	Unmanned Aerial Vehicle
URLLC	Ultra-Reliable Low Latency Communications
VNF	Virtual Network Function
VM	Virtual Machine
VP	Virtual Platform

Executive Summary

This deliverable, titled “D4.2-Resource Elasticity Techniques”, details the design, implementation, and evaluation of resource elasticity techniques within the NANCY B-RAN architecture. The main objectives are to introduce elasticity in both the networking and computational resources. The focus of this deliverable is primarily on scalable and efficient resource management mechanisms with advanced online algorithms.

In Section 2, the deliverable provides an overview of resource elasticity, addressing the challenge of scaling in distributed systems (Section 2.1) and justifying the adoption of Muti-agent deep reinforcement learning (MADRL) to manage computational resources by providing solid theoretical backgrounds of different MADRL-based algorithms (Section 2.2).

Then, Section 3 provides an overview of the NANCY orchestrators that can be controlled by resource elasticity decision engines, namely the Slice Manager (Section 3.1) and Maestro (Section 3.2). These two orchestrators support resource elasticity through their API. They can dynamically adjust resources based on when certain API functions are called, enabling efficient vertical and horizontal scaling. The tradeoff metrics used to monitor the system and trigger scaling are identified in Section 3.3.

The elasticity techniques that can empower the decision engines controlling the orchestrators of the host operating systems are described in Section 4 and constitute the main contributions of this deliverable. The novel elasticity techniques, including SCHED_DEADLINE (Section 4.1), PHaul (Section 4.2), and MADRL-based solutions for computational resource elasticity (Section 4.3), are explained and evaluated in detail. First, SCHED_DEADLINE is used to manage CPU resources by providing predictable CPU allocation to time-sensitive tasks, ensuring that applications with strict latency requirements maintain consistent and fair performance. Second, PHaul, a dynamic path allocation mechanism based on DRL is able to effectively manage network load and enhance the resilience of the network by allocating network resources across paths to meet throughput and fairness constraints. Lastly, MADRL-based computational resource elasticity leverages MADRL algorithms, such as Proximal Policy Optimization (PPO) and Deep Q-Networks (DQN) to dynamically scale CPU and memory resources within network slices, enabling efficient, disruption-free service by responding to fluctuating demand in real time.

1. Introduction

NANCY aims to provide flexible resource management and smart pricing, which are realized in WP4 through the following key objectives:

(a) design low-complexity computational offloading and social-aware caching mechanisms;

(b) identify the B-RAN functions whose operation should be adjusted to the available computational/MEC resources;

(c) characterize the trade-off between NFs performance and resource usage;

(d) develop low-complexity proactive scaling mechanisms;

(e) develop ultra-reliable and low-latency cooperative and multi-hopping access schemes tailored for delay, security, and resilience critical applications, hence addressing the requirement for reliable communication with latency limitations;

(f) develop smart pricing policies that will significantly reduce the ownership cost.

1.1. Relation to Other Tasks and Deliverables

While Objectives (a) and (b) have been addressed in D4.1, and Objectives (e) and (f) will be addressed in D4.3, D4.4, and D4.5, D4.2 focuses on Objectives (c) and (d), which is the outcome of T4.2 titled "Resource elasticity enabling techniques" and will be reported in this deliverable. T4.2 focuses on developing a resource elasticity framework, which is structured around two key pillars: (1) computational elasticity and (2) MEC elasticity. This task develops advanced online algorithms to dynamically and efficiently allocate network and computational resources to users and devices at various time scales. These algorithms aim to ensure Quality of Service (QoS) and Quality of Experience (QoE) for multiple served users and devices, which can be achieved by SLAs (Service Level Agreements) developed in D4.1.

Guided by the parameters defined in the SLA, NANCY leverages AI-driven automation to enable dynamic resource allocation and optimize network performance in real time. This dynamic resource allocation is supported by Network Functions Virtualization (NFV), which enables the flexible deployment of network functions, while network slicing divides the network infrastructure into multiple virtual slices, each slice tailored to specific applications and user needs. These needs are defined in SLAs, which outline the performance guarantees, such as maximal tolerable latency or minimal available bandwidth, that the network must meet to ensure QoS across different slices. To maintain this balance, effective resource management is crucial. AI algorithms and resource elasticity mechanisms work to optimize the allocation of networking and computational resources to adapt to varying demands and provide optimal QoS/QoE for users. This flexible resource adjustment ensures that network functions receive the necessary resources to perform optimally under constrained conditions, preserving QoS and QoE standards.

1.2. Purpose of the Document

In this deliverable, we elaborate on innovative resource elasticity techniques, for time-critical tasks in Section 4.1, multi-hop path allocation in Section 4.2, and computation elasticity technique in Section 4.3. These techniques can be used to power decision engines that interface with existing NANCY orchestrators discussed in Section 3. Moreover, these techniques can be deployed on edge devices and enable MEC elasticity. As the proposed MEC and computational elasticity techniques are based on reinforcement learning, we provide a related theoretical background in Section 2.

Two key enablers in NANCY for resource elasticity techniques, the Slice Manager and Maestro (Section 3) lay the foundation for resource elasticity across diverse demands and network conditions through virtualization technologies, enabling the components introduced in the following paragraphs to

function effectively. More specifically, the existing Slice Manager, serves as a resource orchestrator, controlling network slices via an API gateway and enforcing resource allocation and slice reconfiguration across clusters. Together with the AI Virtualizer, developed in D3.4, the Slice Manager dynamically reallocates inter-slice resources using a multi-agent deep reinforcement learning (MADRL) framework, enhancing efficiency and maximizing resource utilization. Maestro, the service orchestrator in NANCY, manages service lifecycles across geo-distributed infrastructures. By automating deployment, scaling, and real-time resource adjustments, Maestro enables the optimization of the performance and scalability across edge and core networks.

The fluctuating demand of applications, especially those with time-critical tasks, require guaranteed QoS for reliable performance. In NANCY, this is achieved with SCHED_DEADLINE (Section 4.1), a reservation-based scheduler within the Linux kernel. Through Constant Bandwidth Server algorithm with Earliest Deadline First (EDF) scheduling, SCHED_DEADLINE enables predictable CPU allocation by allowing the orchestrator to dynamically adjust parameters in the SCHED_DEADLINE, this approach prevents overuse and ensures fair resource distribution. By supporting real-time vertical scaling, it allows resources to adapt as demands shift, helping NANCY maintain service quality and optimize resource utilization efficiently.

PHaul (Section 4.2), a DRL-based component is deployed to achieve network resource elasticity. It oversees the fine-grained distribution of network resources, ensuring that each slice meets its performance targets, such as throughput, and fairness, without over-provisioning or underutilizing resources. The PHaul design incorporates a Digital Twin, which replicates real-time traffic and network conditions to facilitate dynamic flow allocation. This integration allows the PHaul to intelligently and adaptively select optimal paths for network traffic, ensuring seamless and efficient service delivery across varying network demands.

Computational elasticity is established as one of the pillars of T4.2. It is achieved through multi-agent Deep Reinforcement Learning (MADRL)-based techniques (Section 4.3). These techniques dynamically scale computational resources such as CPU and memory utilization within a slice to ensure disruption-free service. The MADRL framework is particularly effective in environments with fluctuating traffic loads, as it formulates resource allocation as a Markov Decision Process (MDP), enabling continuous learning and adaptation to real-time network conditions. By utilizing DRL agents for each service within the slice, the system can make decentralized decisions regarding resource scaling, thereby optimizing the overall performance. These MADRL-based approaches provide real-time control and management of resources within the slice, allowing the orchestrator to adjust resources dynamically without interrupting services, even in response to unexpected spikes in demand. This deliverable reports on the results of Task 4.2 towards NANCY R10— Experimentally driven reinforcement learning optimization of B-RAN discussed in D3.1.

1.3. Structure of the Document

The rest of the document is structured as follows:

- **Section 2 – Background on resource scaling and MADRL-based scaling** presents the background concerning the resource allocation by leveraging methodologies based on MADRL.
- **Section 3 – Virtualization Platform** documents the main components of the virtualization platform, namely the Slice Manager and Maestro..
- **Section 4 – Novel Resource Elasticity Mechanisms** describes the mechanisms that were developed for allocating and scheduling the available resources.
- **Section 5- Conclusion** summarizes and concludes the deliverable.

2. Background on resource scaling and MADRL-based scaling

2.1 Definition of Resource Scaling

The ever-changing traffic needs in B5G communications networks introduce new challenges, making scalable computing resources crucial for services and applications to meet their demands efficiently and cost-effectively. Managing cloud resources well is essential to maintaining high-quality service levels, preventing resource underutilization, and avoiding system overloads. There are two primary scaling mechanisms: horizontal scaling and vertical scaling, which are used to adjust the network resources to meet demands. These dynamic scaling mechanisms are highly important in B5G networks since they allow the automatic allocation of new resources, namely CPU, memory, and storage resources, during peak usage to maintain user QoS. However, in off-peak periods, they automatically release exceeding resources, enabling 5G network slicing to achieve dynamic balance and automated resource allocation, enhancing the flexibility of the 5G network architecture [1]. For example, the Third Generation Partnership Project (3GPP) defines three service types for network slicing: Ultra-Reliable Low Latency Communications (URLLC), Massive IoT (MIoT) and Enhanced Mobile Broadband (eMBB). Scaling adjusts the capacity of individual slices, hosting network functions and applications (VNFs) in such a way that allows for the most efficient resource exploitation. In addition, 5G core network slicing specifies network services according to the functional and quality requirements of the use cases mentioned above.

Horizontal scaling, or scaling out/in, involves adding or removing entire resource units, such as virtual machines (VMs) or containers. This approach is particularly suited for inter-slicing, where resources are shared across multiple slices to support various use cases. By expanding or contracting resources without disrupting individual slices, horizontal scaling helps balance load across slices, adapting to demand changes while supporting diverse applications on a shared infrastructure.

Vertical scaling, or scaling up/down, increases or decreases resources allocated to an existing instance, allowing smoother adjustments within a single slice to meet specific demands without adding new units. This method is especially relevant for intra-slicing, where resources are optimized within an individual slice. Vertical scaling enables each slice to adjust its resources internally, meeting the unique requirements of its hosted applications or services. For example, a slice handling time-sensitive applications can scale up CPU or memory within that slice to ensure reliable performance during high load periods.

To address the complexity of dynamic scaling and resource optimization across slices, NANCY employs DRL solutions in Section 4.2 and MADRL-based solutions in D3.4 and Section 2. This AI-driven approach enables decision-making across network slices, dynamically adjusting resources to ensure efficient, high-quality service delivery in response to shifting demands.

2.2 Multi-agent Deep Reinforcement Learning (MADRL)-based Scaling

2.2.1 Introduction to Markov Games and Multi-Agent Reinforcement Learning

Markov Games (also known as stochastic games) extend the concept of Markov Decision Processes (MDPs) to a multi-agent setting, providing a formal framework for modelling strategic interactions between multiple decision-makers in a dynamic environment [2]. As a matter of fact, Markov Games represent the evolution of traditional game theory in mathematics, in which the players involved in the game can take only fixed (i.e.: based on predetermined strategy) decisions or actions, thus leading to an overall static shared optimization. With Markov Games, instead, it is possible to add a dynamic dimension to the game, thus making the state evolve based on the decisions made by the players at each time step.

A Markov game can be defined by the following tuple:

$$MG = (N, S, A_i, P, R_i, \gamma)$$

where:

- S is the overall state space, identifying all the measurable quantities that the agents can take from the shared environment. Note that there may be a unique set of states for all the players, or each player may have its own private state space S_i.
- A_i is the action space of agent i, detailing the actions available to each agent.
- $P(s'|s, a_1, \dots, a_N|)$ is the transition probability function, determining the likelihood of moving from state s to state s' given the joint actions.
- $R_i(s, a_1, \dots, a_N)$ is the reward function for player i.
- $\gamma \in [0,1]$ is the so-called discount factor, which weighs the importance of future rewards.

In this concept, several agents, each with potentially opposing or complementary aims, interact with the same (shared) environment. These relationships can be cooperative (e.g., all agents share the same prize) [3], competitive (e.g., zero-sum games) [4], or mixed (e.g., combining cooperation and competition) [5]. Considering not just the dynamics of the environment, but also the decisions and behaviors of other agents, each agent seeks to maximize its own cumulative reward over time. In this setting, the concept of Markov game equilibrium typically refers to a Nash equilibrium, which occurs when no agent can improve its expected cumulative reward by unilaterally changing its policy, assuming the other agents' policies remain fixed [6].

One of the most common strategies to address and solve Markov Games is a subfield of Reinforcement Learning (RL), namely Multi-Agent Reinforcement Learning (MARL). Unlike single-agent RL, where the environment is often believed to be stationary, MARL brings additional complexity because each agent's activity dynamically affects the environment and the other agents' learning processes. Hence, in MARL, each DRL agent corresponds to a specific player and it can perform autonomous actions in the same environment, thus influencing the reward of the other players [7]. MARL presents some inherent difficulties:

- From the standpoint of any one agent, the environment becomes non-stationary as each agent modifies its policy over time.
- To learn about the environment and take advantage of the knowledge it has acquired, each agent has to explore it. This equilibrium is made more difficult in a multi-agent situation due to the existence of additional learning agents.
- Agents must learn to coordinate their behaviors in cooperative situations, thus requiring a communication infrastructure. Layers of complexity are added because agents would have to exchange information in order to coordinate their tactics.
- In competitive circumstances, agents must predict the actions of their opponents, making learning analogous to strategic thinking or game theory.

To account for the presence of numerous agents, MARL algorithms expand classical RL techniques such as policy gradient methods or Q-learning. Although some methods consider other agents as part of the surroundings and ignore their strategic behavior, other sophisticated methods actively simulate other agents' policies or behaviors to enhance learning outcomes.

The disciplines in which Markov games and MARL are applied are numerous and include (i) robotics, where groups of robots may have to coordinate to satisfy some duties; (ii) economics, to describe strategic interactions in markets or negotiations; (iii) video games, in which agents have to cooperate or compete with human players; and (iv) resource management, where multiple agents (e.g., servers, machines, users) need to collaboratively or competitively allocate, distribute, or manage limited resources over time, with the goal of optimizing system-wide or individual performance metrics.

In particular, resource management problems often involve balancing the competing interests of different agents while managing shared, finite resources. MARL provides a powerful framework for addressing these challenges by enabling agents to learn cooperative or competitive strategies in complex, dynamic environments [8].

To account for the presence of numerous agents, MARL algorithms expand classical RL techniques, by defining proper rules for exchanging information about the state space, in the case of cooperative games. To accomplish T4.2 missions, we are going to explore Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO) as RL algorithms applied to multi-agent resource management practices.

With DQN [9], an agent learns an optimal policy by estimating the Q-value function, which calculates the predicted future rewards for actions in various states. DQN is a value-based reinforcement learning method. DQN has been effectively used for numerous RL tasks, including multi-agent scenarios, and it approximates the Q-value function using deep neural networks. In MARL, every agent keeps track of its own Q-network, learning by applying the Bellman equation to update its Q-values. Based on these values, the policy is determined (usually via ϵ-greedy action selection). Its primary benefits stem from the fact that DQN is very efficient in discrete action spaces, which is the situation for most resource management scenarios and can scale well to situations with vast state spaces by utilizing neural networks to approximate Q-values.

On the contrary, PPO [10] is a policy-based algorithm, specifically a sort of policy gradient method. By adjusting a policy network's parameters, PPO directly optimizes the policy as opposed to learning a Q-value function. To guarantee that policy updates are not excessively big, PPO employs a clipped surrogate objective function, which aids in stabilizing training. The clipped objective function of PPO lowers the possibility of abrupt updates, improving convergence qualities and learning stability. PPO's main advantage is that it performs well in continuous action spaces, which increases its adaptability in a larger range of MARL problems. By sampling trajectories over full episodes and using them for learning, PPO promotes exploration and is especially helpful in cooperative multi-agent systems.

The performance of these two RL algorithms will be compared in the next section of the present deliverable, whereas the next subsection will show how tools based on Markov games and MARL can be of great use in solving typical resource management problems.

2.2.2 MADRL-based Solution for Resource Management

As shown in the previous section, MARL extends the traditional RL framework by involving multiple autonomous agents, each with the capacity to learn, adapt, and make decisions independently, while interacting with a shared environment. These agents have the goal of maximizing their individual or collective cumulative reward over time. The relevance of this discipline in the resource management domain lies in its decentralized nature, which allows it to operate effectively in distributed and highly complex environments where centralized control is impractical or inefficient. This is particularly significant in scenarios involving the management of computational, network, and energy resources, such as in modern wireless communication systems, cloud infrastructures, and the Internet of Things (IoT), where various performance metrics, including quality of service (QoS), energy efficiency, and latency, must be optimally balanced to meet ever-increasing demands.

In this sense, the explosive increase in the number of devices, the advent of new high-demand applications, and the emergence of heterogeneous networks (HetNets) pose a significant challenge [11].

Significant issues arise in managing computational and energy resources in distributed systems and networks, which MARL can handle well. Future wireless networks, with their heterogeneity and scalability, will require efficient management of numerous devices with differing computational and energy needs, supporting applications like autonomous vehicles and mobile edge computing. Conventional centralized control techniques have trouble scaling and optimizing in real-time, which

frequently results in inefficient resource allocation and excessive energy use. On the other hand, MARL allows agents to learn and adjust in response to their interactions with the environment, allowing them to optimize tasks like energy consumption and spectrum allocation even in unpredictable and dynamic environments. The decentralized method of MARL provides scalable and adaptable resource management solutions, especially in complex, multi-objective situations, and is well-suited to scenarios in which several agents cooperate or compete.

Several studies have applied MARL to optimize resource management across various domains. One of the areas is resource management in computational systems, particularly in telecommunication environments. In these contexts, dynamic resource allocation is crucial for optimizing the use of CPU, memory, and bandwidth while reducing operational costs. For instance, in edge computing, the authors in [12] propose a Decomposed Multi-Agent DDPG (DMDDPG) framework, where agents distributed across edge nodes collaboratively learn efficient resource allocation policies, leading to improved task execution times and enhanced resource utilization in dynamic environments. Differently, the work in [13] presents a scalable MARL framework for distributed wireless resource management. Each RL agent makes independent, simultaneous decisions regarding user scheduling and power control, showing significant robustness to environmental changes, and achieving performance levels comparable to centralized systems while maintaining scalability.

Furthermore, multi-access edge computing (MEC) powered by unmanned aerial vehicles (UAVs) has emerged as a promising solution for future space-aerial-terrestrial integrated communications. In response, the authors in [14] propose a Federated Multi-Agent Reinforcement Learning (MAFRL) algorithm. This semi-distributed framework jointly optimizes resource allocation, user association, and power control, resulting in a 23% reduction in operation time compared to centralized algorithms.

In the Industrial Internet of Things (IIoT), resource allocation for edge devices is significantly enhanced by MARL. For instance, the work described in [15] integrates Deep Reinforcement Learning (DRL) with multi-agent systems to optimize the allocation of computational resources and bandwidth, aiming to maximize resource efficiency in response to dynamic system changes. This approach effectively reduces network traffic, computational load, and processing time, thereby ensuring minimal resource consumption and improving overall performance, particularly in terms of latency and error rates. Conversely, [16] explores a different angle by employing a Proximal Policy Optimization (PPO)-based MARL algorithm within a blockchain-supported hierarchical digital twin IoT framework. Here, the focus is on optimizing resource allocation for IoT devices by minimizing system delay and energy consumption, while simultaneously ensuring system reliability and learning accuracy. This balance between low-latency communication and energy efficiency highlights the versatility of MARL in managing resources effectively within increasingly complex and resource-constrained environments.

In B5G networks, the rise of interconnected subnetworks presents new challenges, particularly with regard to interference management. To address this, the authors in [17] propose an intelligent radio resource management method based on MARL. This approach simplifies resource management by using only the received signal strength indicator (RSSI) for each channel, thus avoiding the complexity of gathering channel gain measurements, which is a significant advancement for efficient dynamic resource allocation.

The versatility of MARL in addressing resource allocation challenges is evident across various domains, as illustrated by several recent studies.

In the automotive industry, the article [18] explores resource allocation optimization in connected and autonomous vehicles (CAVs). It introduces a Secondary Resource Allocation (SRA) mechanism that utilizes a dual time scale for resource distribution among vehicles. By modelling the service process as a queuing system, each task request is treated as an individual agent, and the MARL algorithm is employed to coordinate resource allocation effectively based on vehicle states and queue conditions. The simulation results reveal a notable 13% increase in task completion rates.

Similarly, the article [19] addresses the intricate challenges of joint spectrum and power allocation within vehicular communication networks, critical for enhancing efficiency in autonomous driving through cooperation between vehicles and infrastructure. The authors introduce a novel methodology as complete-game MARL (CG-MARL), which combines MARL with cooperative stochastic game theory, enhancing stability and scalability in resource allocation as the number of vehicles increases. It also incorporates mean-field game (MFG) theory to reduce computational resource consumption while maintaining near-optimal performance.

In the context of energy management, the article [20] tackles the complexities faced in multi-building multi-energy virtual power plants. The authors present an innovative approach that integrates a multi-agent transformer with a parallel adapter module, facilitating streamlined coordination among building agents through sequential modelling.

Eventually, authors in [21] investigate the challenges associated with resource management in Serverless Function-as-a-Service (FaaS) environments to mitigate tail latency and enhance resource utilization. They highlight the limitations of traditional single-agent reinforcement learning algorithms and propose MA-PPO, a multi-agent reinforcement learning algorithm built on Proximal Policy Optimization (PPO), aimed at improving overall performance.

These articles highlight the broad applicability of MARL in optimizing resource allocation across different sectors. By using multi-agent systems and advanced algorithms, MARL improves efficiency and scalability while addressing complex challenges. This versatility showcases MARL's potential to drive innovation and enhance computational and network resource management in future technologies.

3. Virtualization Platform

3.1 Scaling With Slice Manager

3.1.1. Introduction of Slice Manager

The slice composition workflow in the ETSI framework provides a systematic methodology for the creation and management of network slices, which is essential for ensuring dynamic and efficient network operations. This process is facilitated by a specialized tool known as the Slice Manager (SM), which executes a series of well-defined steps. The workflow commences with (i) the allocation of compute resources and chunks. This step involves integrating a Kubernetes (K8s) cluster into the SM using its Kubeconfig file, validating connectivity and credentials, and subsequently retrieving and storing relevant resource information in the database. Each compute chunk corresponds to a K8s namespace configured with specific allocations of computing resources, memory, and storage. The next phase (ii) addresses the setup of network resources and chunks, which ensures that the network slice possesses the necessary connectivity infrastructure, typically in the form of a VLAN. Following this, (iii) radio resources and chunks are established, specifying the required wireless functionalities, such as physical resource blocks (PRB) and cell frequencies (ARFCN). Once all resources are configured, the process proceeds to (iv) slice creation, where these discrete chunks are logically integrated into a cohesive network slice through their associated chunk IDs. Upon successful slice creation, the workflow transitions to (v) activation, making the slice fully operational and ready for deployment. The final step (vi) involves application instantiation within the slice, which requires onboarding the application on the Open Source MANO (OSM) platform. This step enables the provision of specific services and functionalities aligned with the network's requirements. This structured workflow ensures that all elements of the network slice are meticulously planned, integrated, and executed, thus supporting robust and scalable network management solutions.

3.1.2. Scaling API

The Slice Manager enables up/down scaling computing, memory, and storage resources dedicated to an active network slice by modifying the Edge/Cloud chunk quotas through PUT calls, as shown in Figure 1.

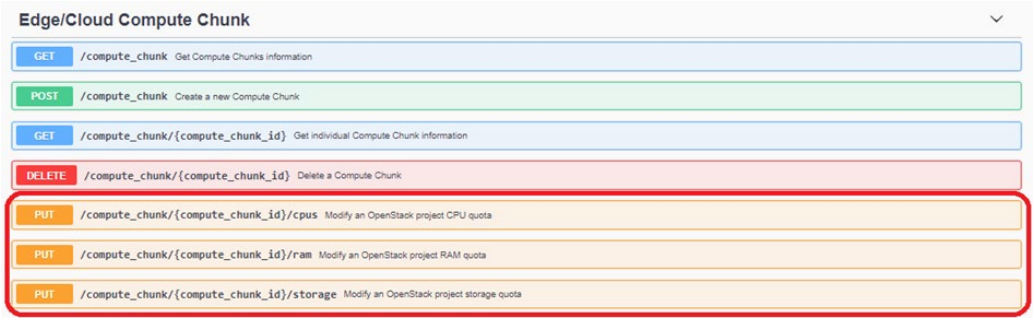


Figure 1: PUT calls for enabling scale up/down resources in Slice Manager API.

The calls' parameters are provided in this excerpt from the API. Yaml, as presented in Figure 2.

```
/compute_chunk/{compute_chunk_id}/cpus:
  put:
    tags:
      - Edge/Cloud Compute Chunk
    summary: Modify a K8s project CPU quota
    description: >-
      K8s Project CPU quota modify method
    operationId: modifyCpuComputeChunk
    parameters:
```

```
    - in: path
      name: compute_chunk_id
      required: true
      schema:
        type: string
    requestBody:
      description: The body of the request
      required: true
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/ComputeChunkNewCPUInput"
    responses:
      '204':
        description: Request succeeded
      '404':
        $ref: '#/components/responses/NotFound'
      '501':
        $ref: '#/components/responses/NotImplemented'

  /compute_chunk/{compute_chunk_id}/ram:
    put:
      tags:
        - Edge/Cloud Compute Chunk
      summary: Modify a K8s project RAM quota
      description: >-
        K8s Project RAM quota modify method
      operationId: modifyRamComputeChunk
      parameters:
      - in: path
        name: compute_chunk_id
        required: true
        schema:
          type: string
      requestBody:
        description: The body of the request
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/ComputeChunkNewRAMInput"
      responses:
        '204':
          description: Request succeeded
        '404':
          $ref: '#/components/responses/NotFound'
        '501':
          $ref: '#/components/responses/NotImplemented'

  /compute_chunk/{compute_chunk_id}/storage:
    put:
      tags:
        - Edge/Cloud Compute Chunk
      summary: Modify a K8s project storage quota
      description: >-
        K8s Project Storage quota modify method
      operationId: modifyStorageComputeChunk
      parameters:
      - in: path
        name: compute_chunk_id
        required: true
        schema:
          type: string
      requestBody:
        description: The body of the request
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/ComputeChunkNewStorageInput"
      responses:
```

```
'204':
  description: Request succeeded
'404':
  $ref: '#/components/responses/NotFound'
'501':
  $ref: '#/components/responses/NotImplemented'
```

Figure 2: Parameters from the API

3.2 Scaling With Maestro

Maestro is a prototype service orchestrator for managing the lifecycle of end-to-end services atop geo-distributed heterogeneous infrastructures.

Today's infrastructures expand towards the end users, where numerous Internet of Things (IoT) and/or user equipment (UE) devices require connectivity with local (edge) or remote (far in the cloud) services. This connectivity may often be provided via cellular (5G and beyond) networks or low-power IoT networks integrated with multiple geo-distributed (edge and core) cloud infrastructures.

3.2.1. Introduction of Maestro Orchestrator

Maestro is a cloud-native service orchestrator that provides automated service deployment, localization, lifecycle management, and scaling, while dynamically exploiting the underlying network services for optimizing service performance, security, and scalability. During runtime, Maestro also implements a policy framework that associates service instances with high-level policies.

Such complex ecosystems pose several challenges in the way service onboarding, deployment, and lifecycle management (LCM) are performed in an end-to-end fashion, mainly because:

- Modern services are structured as collections of independently deployable and loosely coupled microservices.
- Modern services may span across multiple administrative domains - managed by different owners - who often do not trust each other.
- Modern services are often associated with service level agreements (SLAs), which pose strict compute (i.e., CPU, main memory, storage) and network (i.e., latency, throughput, packet loss) requirements.

Maestro is a holistic end-to-end service orchestration platform that aims to bridge these gaps by offering zero-trust multi-domain service orchestration abstractions to various stakeholders. Specifically, Maestro addresses requirements for the following stakeholders:

Infrastructure Owners:

- Allows infrastructure owners/providers to register new (private) domains and services under the platform's realm through a programmable zero-trust connectivity (ZTC) fabric.
- Manages services across multiple geo-distributed "non-trusted" domains acting as the root of trust.

Service Providers:

- Allows service providers to package distributed services as if they are centralized, thus moving complexity to the platform.
- Offers a single set of service management APIs, no matter how many domains an application expands to.
- Supports state-of-the-art service packaging tools (i.e., Kubernetes, helm, and docker-compose).

- Provides programmable connectivity services across clusters/domains, giving a single-cluster illusion to the users.
- Provides a real-time view of the deployed service instances' state.
- Provides knobs to change a service instance's runtime state via a service update API and/or real-time policies.

Relevant Platform Providers:

- Decouples service and resource management via integration with operations support systems (OSS) using open standardized APIs. In the NANCY project, Maestro comes with ETSI OpenSlice (OSL SDG) only when it comes to computing resources.
- Maestro deals with service management.
- Delegates resource management to OSS (e.g., ETSI OpenSlice).
- Manages multiple OSS instances.
- Integrates with vanilla container orchestration platforms (i.e., Kubernetes).

3.2.2. Internal Architecture, Technologies, and Baseline Assets

Figure 2 depicts a high-level functional architecture of Maestro that will be used as a baseline platform for NANCY. At the northbound API, Maestro expects service providers to package their services in one or more containers forming a service graph. Once a containerized service is available, Maestro offers a UI (and a northbound API) that allows service providers to onboard the containerized service in an intuitive manner (step 1 in Figure 2). In step 2, a complete service is declared in Maestro's language, and service providers can order an instance of this service. This requires Maestro to create a service-level slice (step 3 in Figure 2), formulate a slice intent message towards a specific OSS (step 4 in Figure 2), and dispatch this slice intent message to the underlying OSS (step 5 in Figure 2).

Note that steps 3-5 are necessary for a service deployment as Maestro is a service-level orchestrator and, thus does not have a direct view of the underlying infrastructure (only infrastructure-level does have such view). For this reason, Maestro requests a certain amount of computing and network resources (i.e., a slice) to be allocated by an OSS, on top of which Maestro performs service deployment. When the underlying OSS allocates the requested slice (step 6 in Figure 2), the slice is returned to Maestro (step 7 in Figure 2) and service deployment begins. In this step, Maestro takes control of the allocated slice by connecting to the designated endpoints of the virtual infrastructure to initiate service deployment (step 8 in Figure 2). Maestro allows service providers to deploy their services atop both Infrastructure as a Service (IaaS) platforms, such as OpenStack and Platform as a Service (PaaS) platforms, such as Kubernetes. Maestro's deployment engine spawns the appropriate containers in the case of Kubernetes or Virtual Machines (VMs) in the case of IaaS platforms and requests the monitoring module to deploy its monitoring routines (step 9 in Figure 2). Finally, during service runtime, Maestro invokes a dedicated LCM component for managing the lifecycle of deployed service instances, as shown in step 10 in Figure 3.

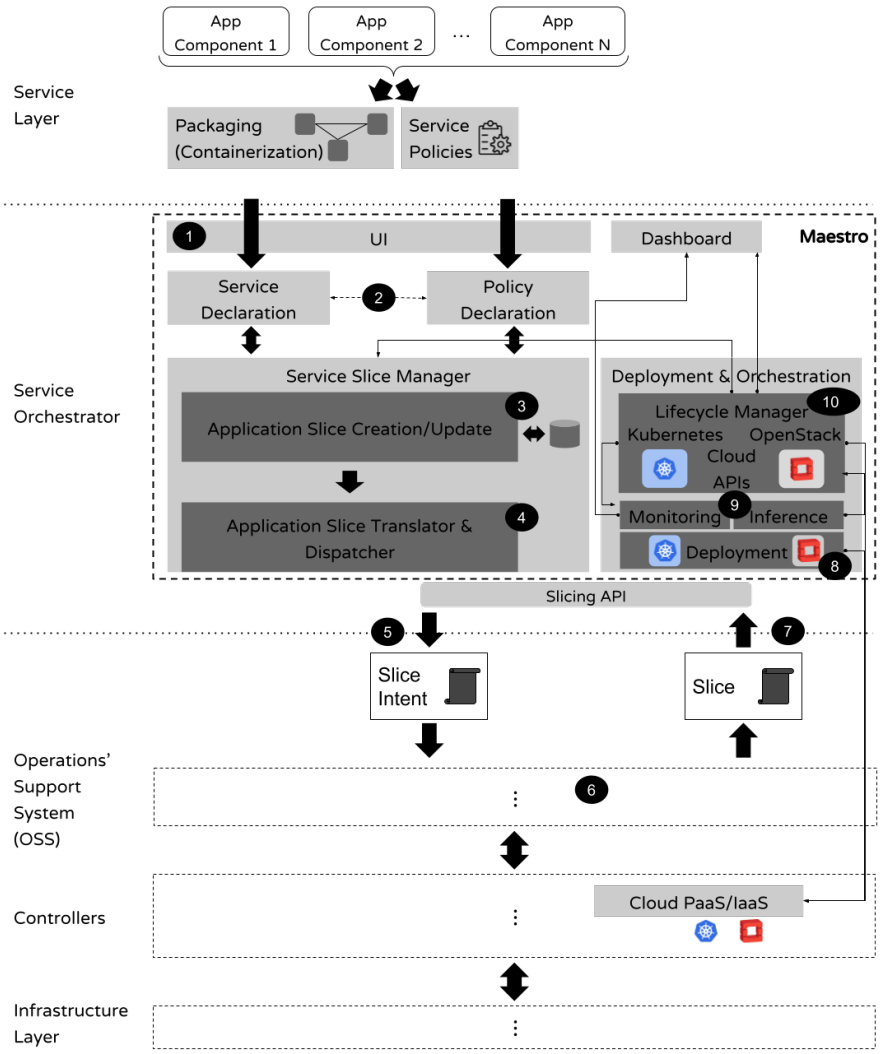


Figure 3: Maestro's high-level architecture.

3.2.3. Functionalities

Table 1 presents the key functional requirements of Maestro for a successful integration with the rest of NANCY's ecosystem.

Table 1: Maestro's functional requirements.

FR-ID	FR-Description	Related Module(s)/ Component(s)
FR1	Access to the NANCY services repository through an API	NANCY services repository
FR2	Offer NANCY end user Service Exposure set of APIs through TM Forum's service catalogue API (northbound/user facing API)	End Users
FR3	Support TM Forum's service order and inventory APIs in collaboration with OpenSlice (southbound API)	OpenSlice
FR4	Manage cloud applications through an API on each testbed	Testbeds
FR5	Manage telemetry agents per application component	Testbeds
FR6	Consume telemetry in Prometheus format	Testbeds

3.2.4. External APIs

Maestro requires several external interfaces that enable communication with adjacent components for service orchestration, data collection, and consumption of reusable artifacts. These interfaces were presented in Deliverable *D6.1 - B-RAN and 5G End-to-end Facilities Setup* (Section 4, Table 4-1), and include:

- The end user Service Exposure (BSS) set of APIs through NIS1
- The Resource Service Exposure set of APIs through NIS2
- The NANCY Service Repository and Registry set of APIs through NIS3

3.2.5. Maestro Architecture in NANCY

The final form of NANCY's end-to-end service orchestrator (Maestro[1]) is depicted in Figure 4. At the northbound, Maestro exposes a set of standardized open APIs based on TMForum, to facilitate interaction with stakeholders (e.g., end-users, service providers, etc.) or peering systems (e.g., an operations support system (OSS), such as ETSI OpenSlice[2])

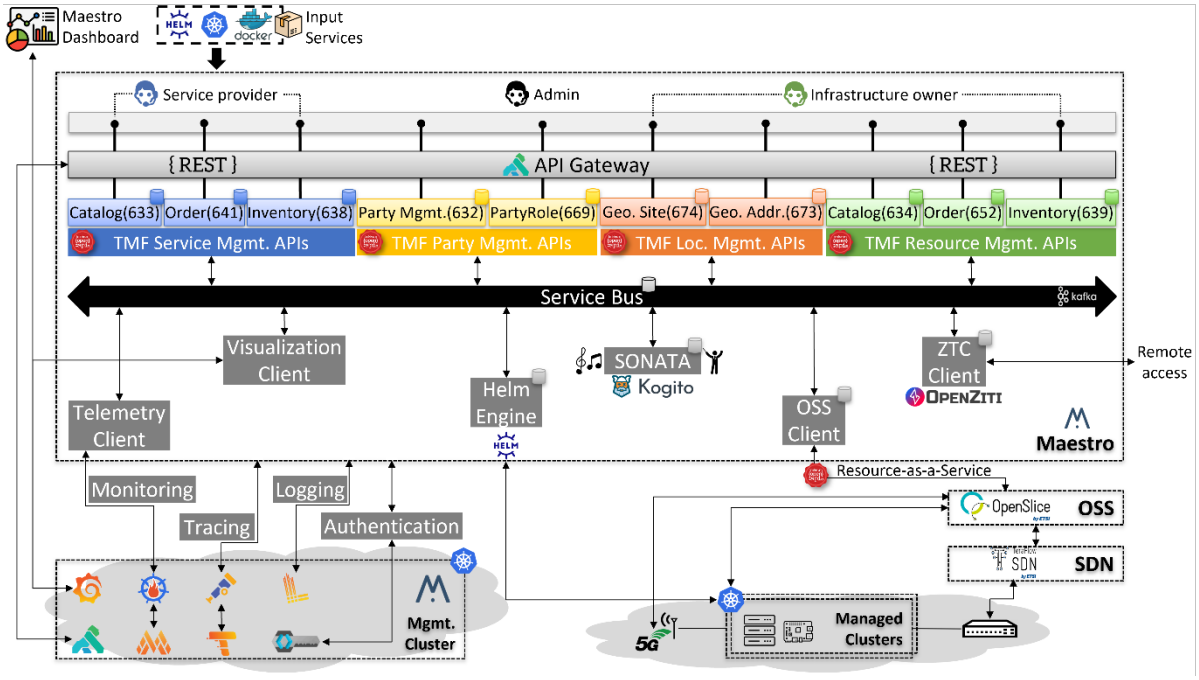


Figure 4: NANCY End-to-end Service Orchestrator (Maestro).

The internal components of the Maestro orchestrator are summarised in Table 2, where relevant descriptions and useful references are provided for further clarification.

Table 2: Maestro software modules used in NANCY

Software module	Description
API Gateway	A thin entry-point to the Maestro TMF APIs – based on Kong[3] – which dispatches input requests to the correct TMF API endpoint.
TMF API	The entire set of TMF APIs supported by Maestro are highlighted in different colours according to Figure 3 as follows:

[1] UBITECH, "Maestro open documentation," [Online]. Available: https://maestro-mkdocs.readthedocs.io/
[2] ETSI, "OpenSlice (OSL) Software Development Group (SDG)," [Online]. Available: https://osl.etsi.org/
[3] Kong Inc., "Kong Gateway: Simplify API Management. Unlock AI Innovation," [Online]. Available: https://konghq.com/

	(i) onboarding services into various catalogs and categories based on TMF 633 Service Catalog Management API [22]. (ii) ordering services from the catalog(s) based on TMF 641 Service Ordering Management API [23]. (iii) observing the service instances' lifecycle based on TMF 638 Service Inventory Management API [24]. (iv) onboarding resources into various catalogs and categories based on TMF 634 Resource Catalog Management API [25]. (v) ordering resources from the catalog(s) based on TMF 652 Resource Ordering Management API [26]. (vi) observing the resource instances' lifecycle based on TMF 639 Resource Inventory Management API [27]. (vii) managing Maestro stakeholders (both individuals and organisations) using the TMF 632 Party Management [28] and TMF 669 Party Role Management API [29]. (viii) managing the points of presence of infrastructure resources (available locations where Maestro can order services from OpenSlice) using the TMF 674 Geographical Site Management [30] and TMF 673 Geographical Address Management API [31].
SONATA	The heart of Maestro; SONATA employs RedHat's Kogito[4] open-source Business Process Model and Notation (BPMN) software library to encode service orchestration workflows into BPMN diagrams that describe the exact lifecycle of service specifications, service orders, and service instances.
Helm Engine	The component that undertakes to translate every user-defined service into deployable service descriptors that can be applied to an underlying (set of) cluster(s) via Helm[5].
OSS Client	The component that undertakes to interact with one or more OpenSlice[6] instances to order resources for the end-user services [32]. To allocate compute resources, Maestro consumes a Kubernetes-as-a-Service (K8s-aaS) service from OpenSlice, while for 5G-enabled services Maestro consumes a 5G-aaS service from OpenSlice (5G-aaS not available for NANCY).
ZTC Client	The component that undertakes to interconnect Maestro with remote – private (sitting within a restricted domains) – testbeds where Maestro is requested to deploy services. This component is based on OpenZiti zero-trust platform[7]. For example, this component provides Maestro access to private Kubernetes[8] clusters allocated by OpenSlice across NANCY testbeds.
Telemetry Client	A thin component – based on Prometheus[9] – that undertakes to register service telemetry data. This data is useful for service observation, maintenance, and policy execution. Integrates with AI & Analytics.
Visualization Client	A thin component that undertakes to visualize service telemetry data for Maestro stakeholders. This component is likely to be useless for NANCY as the

[4] RedHat, "Kogito Cloud-Native Business Automation", Available: https://kogito.kie.org/
[5] Cloud Native Computing Foundation, "Helm: The package manager for Kubernetes", Available: https://helm.sh/
[6] ETSI, "OpenSlice (OSL) Software Development Group (SDG)," [Online]. Available: https://osl.etsi.org/
[7] NetFoundry, "OpenZiti: Open-Source Zero-Trust Platform," [Online]. Available: https://openziti.io/
[8] Cloud Native Computing Foundation, Kubernetes, "Production-Grade Container Orchestration," [Online]. Available: https://kubernetes.io/
[9] Cloud Native Computing Foundation, "Prometheus: From metrics to insight," [Online]. Available: https://prometheus.io/

	service telemetry will be visualized through the NANCY AI & Analytics component.
Database	A PostgreSQL[10] database that persists the entire TMF schema (service, party, geographic site/address, resource schema) as well as state information of core Maestro components (i.e., SONATA, Helm Engine, OSS Client, and ZTC Client).
Authentication	A Keycloak[11] instance is used to authenticate users against Maestro, before allowing these users to access the Maestro Northbound APIs.
Logging	A central component – based on Grafana Loki[12] – that aggregates, stores, and queries log entries from every Maestro microservice. It integrates well with Grafana[13] and Prometheus[14] allowing to visualize logs and generate alerts out of these logs respectively.
Tracing	A central component – based on Grafana Tempo[15] – that integrates with popular open-source tracing protocols, such as OpenTelemetry[16], to collect and persist traces from every Maestro microservice. It integrates well with Grafana[13] and Prometheus[14] allowing to visualize traces and generate alerts out of these traces respectively.
Monitoring	An open-source cloud-hosted monitoring and alerting system – based on Prometheus[14] – that is enhanced with long-term persistence and high-availability features using Grafana Mimir[17].
Dashboard	An external dashboard that leverages Grafana[13] to integrate with (i) Grafana Loki[12] for visualizing service logs, (ii) Grafana Tempo[15] for visualizing service traces, and (iii) Grafana Mimir[17] for visualizing service SLAs.
Message Bus	A central component – based on Apache Kafka[18] – to enable asynchronous message exchange and event management among all Maestro microservices.

3.2.6. Final Integration Endpoints

Maestro supports and implements the endpoints summarized in Table 3, for more details about the NANCY Interface Set (NIS) and NANCY Interface (NI) as well as for a visual representation of these interfaces with the Functional and deployment view of the NANCY architecture annotated with interface IDs, see D6.1 (Figure 2-1 and Table 4-1.).

[10] PostgreSQL, "The World's Most Advanced Open-source Relational Database," [Online]. Available: https://www.postgresql.org/

[11] Cloud Native Computing Foundation, "Keycloak: Open-source Identity and Access Management," [Online]. Available: https://www.keycloak.org/

[12] Grafana Labs, "Grafana Loki: Log monitoring for faster troubleshooting at scale," [Online]. Available: https://grafana.com/oss/loki/

[13] Grafana Labs, "Grafana: Visualize your data, optimize your performance," [Online]. Available: https://grafana.com/oss/grafana/

[14] Cloud Native Computing Foundation, "Prometheus: From metrics to insight," [Online]. Available: https://prometheus.io/

[15] Grafana Labs, "Grafana Tempo: Distributed tracing system for better application performance," [Online]. Available: https://grafana.com/oss/tempo/

[16] Cloud Native Computing Foundation, "OpenTelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability," [Online]. Available: https://opentelemetry.io/

[17] Grafana Labs, "Grafana Mimir: Open-source, horizontally scalable, highly available, multi-tenant TSDB for long-term storage for Prometheus," [Online]. Available: https://grafana.com/oss/mimir/

[18] Apache, "Kafka: Open-source distributed event streaming platform," [Online]. Available: https://kafka.apache.org/

Table 3: NANCY interfaces implemented by Maestro

Interface-ID	Related Modules	Type
NIS1	Maestro, BSS	Service exposure to NANCY stakeholders
NIS2	Maestro, OSS (OpenSlice)	Service management API
NIS3	Maestro, CI/CD Platform, OSS (OpenSlice)	Service artefacts management API
NIS5	Maestro, OSS (OpenSlice), Telemetry, AI, Analytics	Telemetry and monitoring API
NIS6	Maestro, OSS (OpenSlice), Compute controllers	Service deployment API

Maestro implements the above interfaces as follows:

- **NIS1** - *Service Exposure to NANCY Stakeholders*: integration is performed via a production Maestro swagger API dedicated to NANCY-related services. Table 5 shows the endpoints that Maestro exposes to NANCY stakeholders via NIS1.
- **NIS2** - *Service Management API with OSS (OpenSlice)*: Through this interface, Maestro communicates with the OpenSlice Operations Support System (OSS) to manage service deployment. By leveraging OpenSlice's capabilities, Maestro can request the allocation of necessary compute resources, supporting orchestrated deployments without directly interacting with the infrastructure.
- **NIS2** - *Service Management API with OSS (OpenSlice)*: Through this interface, Maestro communicates with the OpenSlice Operations Support System (OSS) to manage service deployment. By leveraging OpenSlice's capabilities, Maestro can request the allocation of necessary compute resources, supporting orchestrated deployments without directly interacting with the infrastructure.
- **NIS3** - *Service Artefacts Management API*: This interface facilitates interactions between Maestro, the CI/CD platform, and OSS, enabling seamless management of service artifacts. It ensures that the necessary service components and configurations are available and updated as needed across the deployment lifecycle.
- **NIS5** - *Telemetry and Monitoring API*: Maestro integrates with monitoring systems such as Prometheus, allowing it to collect and register service telemetry data. This interface is crucial for real-time observation and policy-driven adjustments, maintaining optimal performance and supporting analytic insights.
- **NIS6** - *Service Deployment API with Compute Controllers*: Finally, Maestro utilizes this interface to execute service deployment commands, translating user-defined service descriptors into Helm-based deployment specifications. This enables scalable service orchestration across various platforms, including Kubernetes clusters, which is essential for flexible deployment on cloud or edge infrastructures

Table 4: Endpoints exposed by Maestro

Endpoint	Title	Description	Version
/tmf-api/service-catalog-management/v4	633 Service Catalog Management	Provides a catalog of services	4.0.0
/tmf-api/service-category-management/v4		Provides a category of services that belongs to a certain catalog	4.0.0
/tmf-api/service-candidate-management/v4		Provides a candidate of services that belongs to a certain category	4.0.0
/tmf-api/service-specification-management/v4		Provides a specification of services that maps to a certain candidate	4.0.0

/tmf-api/service-order-management/v4	641 Service Order Management	Provides the ability to query and manipulate active service instances	4.1.0
/tmf-api/service-inventory-management/v4	638 Service Inventory Management	Provides the ability to query and manipulate active service instances	4.0.0
/tmf-api/resource-catalog-management/v4	634 Resource Catalog Management	Provides a catalog of resources	4.1.0
/tmf-api/resource-category-management/v4		Provides a category of resources that belongs to a certain catalog	4.1.0
/tmf-api/resource-candidate-management/v4		Provides a candidate of resources that belongs to a certain category	4.1.0
/tmf-api/resource-specification-management/v4		Provides a specification of resources that maps to a certain candidate	4.1.0
/tmf-api/resource-order-management/v4	652 Resource Order Management	Provides the ability to manage resource orders that comprise of one or more resource specifications	4.0.0
/tmf-api/resource-inventory-management/v4	639 Resource Inventory Management	Provides the ability to query and manipulate active resource instances	4.0.1
/tmf-api/party-management/individual/v4	652 Party Management API	Manage individual parties	4.0.0
/tmf-api/party-management/organisation/v4		Manage corporate parties (i.e., organizations)	4.0.0
/tmf-api/party-role-management/v4	669 Party Role Management API	Manage party roles	4.0.0
/tmf-api/geographic-site-management/v5	674 Geographic Site Management API	Manage resource locations at abstract sites, where each site may contain a list of geographic addresses	5.0.0

3.2.7. Degrees of Freedom of the Slice Manager

The degrees of freedom in resource scaling and allocation, including vertical scaling and intra-slice resource elasticity, are actually implemented by **OpenSlice** within the Maestro framework. While Maestro acts as the **service orchestrator**, coordinating the lifecycle of services, it **delegates the direct management of resources** (like compute, memory, and storage) to **OpenSlice**, which functions as the underlying resource orchestrator.

In this setup:

- **Maestro** handles high-level service orchestration, policy enforcement, and interaction with external components (e.g., via APIs for monitoring and telemetry).
- **OpenSlice** manages resource provisioning at the infrastructure level (like Kubernetes clusters), enabling Maestro to apply vertical scaling, resource elasticity, and specific deployment requirements as defined in the NANCY project.

Thus, Maestro leverages OpenSlice's capabilities to perform the actual resource adjustments required by service instances.

3.3. Metrics for Characterizing Trade-Offs

The trade-off between efficiency and performance in virtualized computing environments is closely tied to CPU and memory metrics such as limits, usage, and utilization, along with the application's response time. For example, high CPU utilization can indicate efficient resource usage, but when utilization approaches or exceeds limits, the environment, such as a slice, may begin throttling the system. This throttling leads to a decrease in resource availability, causing degraded performance and an increase in application response times. Therefore, balancing CPU limits and utilization is crucial to maintaining performance without overloading the system, ensuring that applications remain responsive while maximizing efficiency. Similarly, in network environments, the balance between

efficiency and resilience relies on two critical metrics: throughput and fairness. These metrics are essential for managing network load effectively while ensuring stable performance and equitable resource distribution across network paths. Maximizing throughput reflects effective resource usage across paths, but without fairness, certain paths can become congested, leading to performance degradation. Ensuring fairness alongside throughput prevents bottlenecks and maintains consistent service levels across the network. Below are the several metrics for characterizing such trade-offs that are used in Section 4:

- Response Time: This metric tracks how long the application takes to respond to a request. It's a direct reflection of performance from the user's perspective. As CPU and memory resources become constrained, response times generally increase due to throttling or resource exhaustion.
- CPU Utilization: CPU Utilization measures how much of the allocated CPU resources are being used by the application. High utilization often indicates efficient resource use but can lead to throttling if the demand exceeds the allocated limits. This results in degraded performance, visible through longer response times. Maintaining a balance is crucial to avoid over-provisioning while preventing throttling.
- Memory Utilization: Similar to CPU, memory utilization tracks how much memory the application is consuming. Applications that exceed their memory limits can face performance penalties, such as garbage collection or out-of-memory (OOM) kills, which severely impact response times and stability.
- CPU/Memory Limits: These represent the upper bound of resources the Kubernetes scheduler allows the pod to consume. When utilization approaches or exceeds these limits, the system may begin throttling, reducing performance. Reinforcement learning in the setup helps adjust these limits dynamically based on historical data and current usage trends to balance efficiency and performance.
- Available CPU/Memory in the Environment: This tracks the remaining capacity in the application environment. Monitoring the available CPU and memory ensures that scaling decisions are made with an awareness of application-level resource availability.
- Throughput: This metric measures the volume of data successfully transmitted across a network path within a given time frame. High throughput signifies efficient resource use, which contributes to overall network performance. However, optimizing throughput alone can lead to uneven path utilization, potentially causing congestion on overused paths. Balancing throughput across all paths ensures a smooth data flow without overloading specific segments.
- Fairness: Fairness evaluates the equitable distribution of network resources across paths, preventing any single path from becoming overly congested. This metric is critical in maintaining resilience, especially under varying traffic conditions, as it ensures consistent performance across the network by distributing load evenly.

4. Novel Resource Elasticity Mechanisms

In this section, we elaborate on innovative resource elasticity techniques for time-critical tasks in Section 4.1. Moreover, we provide a detailed explanation of the multi-hop path allocation algorithm, PHual in Section 4.2, which efficiently manages the spectral resource. Finally, the computation elasticity technique is presented in Section 4.3.

4.1 Scheduling for Time-Critical Tasks

In NANCY, the scheduling of time-critical tasks relies on the SCHED_DEADLINE scheduler of Linux [33]. SCHED_DEADLINE is a reservation-based scheduler that allows the provision of QoS guarantees to applications that implement the Constant Bandwidth Server reservation algorithm [34], which in turn relies on the Earliest Deadline First (EDF) scheduling algorithm. SCHED_DEADLINE is an integral part of the Linux mainline kernel, hence available to all Linux users. SCHED_DEADLINE allows reserving a *fraction of the CPU bandwidth under bounded service latency* by configuring two parameters: a budget Q (also called runtime) and a period P. SCHED_DEADLINE ensures that, every P time units, Q time units of execution time budget are provided to the target application. The target application can consist of Linux threads and QEMU/KVM virtual machines. Containers are also compatible thanks to an out-of-tree patch [35], which has been extended in the context of NANCY to be used with newer kernel versions. SCHED_DEADLINE is not only a resource partitioning mechanism: it also acts as a resource enforcer, ensuring that no more than Q time units are allocated every period P. This allows multiple, untrusted, applications to be collocated within the same computing infrastructure (e.g., CPU cores). Parameters Q and P are equivalently mapped to two other parameters: the CPU bandwidth assigned to the reservation $\alpha = Q/P$ and the worst-case service latency $\Delta = 2 \cdot (\mathrm{P} - \mathrm{Q})$ with which the target application can be provided access to the CPU.

Scaling SCHED_DEADLINE reservations. This section describes how to flexibly manage the "vertical" scaling of virtualized applications using SCHED_DEADLINE. Generally speaking, vertical scaling refers to adding more computational capabilities to the current machines. In this context, it refers to adding computational power to SCHED_DEADLINE reservations, which involves: (i) increasing the CPU bandwidth α and/or (ii) reducing the service latency Δ. This has a direct correspondence to finding the most suitable values for the period P and budget Q parameters of reservations and dynamically adapting the parameters when the workload conditions change. Indeed, assigning proper configuration parameters to reservation is of key importance since:

1. If the budget is too small, or the period is too large, QoS constraints cannot be guaranteed;
2. If the budget is too large, or the period is too small, the physical edge platform can be underutilized.

Both conditions are clearly unwanted. Hence, a coarse, generous, provisioning of the parameters, e.g., assigning a large CPU bandwidth to an application without properly tailoring it with its computational needs, would conflict to avoid the platform underutilization. Therefore, proper methods to dynamically detect and monitor application's needs are required.

In D3.4, we addressed the problem of monitoring the runtime budget of SCHED_DEADLINE reservations to avoid resource underutilization. In this deliverable, we complement the budget monitoring mechanism provided in D3.4 by (1) addressing the problem of setting a proper period parameter for SCHED_DEADLINE reservations, thus providing mechanisms to set both the budget and the period of SCHED_DEADLINE reservations and (2) discussing how the runtime monitor (used to

estimate the budget) and period estimator can be used together to achieve a comprehensive vertical scaling of a virtualized application using SCHED_DEADLINE.

4.1.1 Period Estimation

Problem Modeling. The considered system is composed of a distributed network of edge nodes. Each edge node is a (Linux-based) computing platform with homogeneous physical cores. On each edge node serves a set $\mathcal{V}$ of VPs, denoted as v_j, which includes m_j vCPUs $c_{j,1}, \dots c_{j,m_j}$.

A VP can be a virtual machine managed by KVM/QEMU or a container. In the following, we discuss how to match SCHED_DEADLINE reservations with VPs.

Each VP v_j is characterized by a tuple $(\overline{Q}_j, \overline{P}_j)$, where $\overline{Q}_j = \left(Q_{j,1}, \dots Q_{j,m_j}\right)$ and $\overline{P}_j = \left(P_{j,1}, \dots P_{j,m_j}\right)$ are the vectors of budgets and periods of each individual vCPU $c_{j,x} \in v_j$.

The virtual platforms in set $\mathcal{V}$ are scheduled by the Linux operating system according to the EDF algorithm.

The workload running inside each vCPU is scheduled with one of the other Linux schedulers (e.g., the fixed-priority scheduler). Each v_j serves a workload composed of a set of tasks Γ_j. Each task $\tau_i \in \Gamma_j$ is characterized by an execution time C_i and an activation period T_i, meaning that the task is considered releasing a (potentially, infinite) sequence of instances (called jobs), each one spaced by T_i time units.

Estimating the period of tasks. To properly set the reservation scheduling parameters, it is paramount to accurately estimate the tasks' activation patterns. In particular, it is essential to identify tasks that can be modeled through periodic activation patterns and to estimate their activation periods. This can be performed by identifying tasks' activation events and performing a frequency-domain analysis on them [36].

The Linux kernel's *function tracer* (ftrace[19]) is used to extract the tasks' "wakeup" events, indicating that a process or thread becomes selectable by the kernel CPU scheduler, moving from a blocked state to the ready state. The sequence of wakeups for each relevant task is registered by modeling the j^{th} wakeup of task τ_i, occurring at time $r_{i,j}$, as a Dirac delta $\delta(t - r_{i,j})$ centered at time $r_{i,j}$. After collecting N of these events, a function $a_i(t) = \sum_j \delta\left(t - r_{i,j}\right)$ describing the activations of task τ_i is built and is transformed to the frequency domain:

$$\mathcal{F}(a_i(t)) = \int_{-\infty}^{\infty} a_i(t) e^{-j2\pi f t} dt = \\ = \int_{-\infty}^{\infty} \sum_{j=1}^{N} \delta(t - r_{i,j}) e^{-j2\pi f t} dt = \sum_{j=1}^{N} e^{-j2\pi f r_{i,j}}$$

The energy of this Fourier transform is then computed as

$$S_i(f) = \sum_{j=1}^{N} \sqrt{\cos^2(2\pi r_{i,j} f) + \sin^2(2\pi r_{i,j} f)}$$

Identifying peaks in this energy function can then estimate the task's periodicity [27].

[19] The Linux Fundation, ''Kernel's function Tracer',' [Online]. Available: https://www.kernel.org/doc/html/latest/trace/ftrace.html.

The program originally used to detect periodic tasks with a top-like interface [37] has been modified turning it into *PeriodWiz*, (the Period Wizard) [38], a daemon that sets up *ftrace* for tracing the wakeup events of the monitored real-time tasks;

- stores the functions $a_i(t)$ describing the activation patterns of such tasks;
- periodically (with a configurable period T) computes the energy $S_i(f)$ for each monitored task τ_i, looking at the peaks in $S_i(f)$ and identifying the periodic tasks with their periods.

PeriodWiz exports an RPC interface that allows clients to add new tasks to the set of real-time tasks to be monitored (by registering a new process ID) and to query for the period of a monitored task.

Clearly, it is essential to compute the Fourier transform of $a_i(\ \ )$ after collecting an appropriate number N of events: if a too-small number is selected, not enough samples are registered in $a_i(t)$ and the period estimation risks to be based on noisy data; on the other hand, if N is too large, we risk detecting periodic tasks with a too-long delay.

Some experiments about this will be reported in the following. To address this issue, the daemon starts by computing $S_i(f)$ based on a small number of samples and increases N if the task is not identified as periodic. When a maximum N^{max} is reached without identifying a period, the task is marked as "not periodic" and N is not increased further.

Another parameter is the period T of PeriodWiz. It has no influence on the sampling frequency of wakeup events, which are registered by the tracing facilities. However, it is important to tune it properly: if T is too long, a period estimate can be available after too much time and after enough samples have been collected, while if T is too short, PeriodWiz can spuriously wake up (causing overhead at the operating system level) without enough samples being collected.

From task periods to reservation periods. Classical real-time systems consider the parameters of each thread to be known and a static workload (no new thread joins at runtime), enabling the design of the vCPU parameters offline, at design time. The considered edge computing context is instead much different and provides a dynamic workload of VM or containers, which need to correspond to a VP v_j. However, deciding the parameters $Q_{j,x}$ and $P_{j,x}$ is a hard task when no prior information about the workload is available.

The tool PeriodWiz presented in this paper [38] allows to detect the periodicity of a task running in a Linux system.

However, once the periods T_i of the tasks $\tau_i \in \Gamma_j$ assigned to a VP v_j have been obtained, they must be used to configure the VP itself, i.e., the budgets $\left(Q_{j,1}, \dots Q_{j,m_j}\right)$ and periods $\left(P_{j,1}, \dots P_{j,m_j}\right)$, and possibly also the number of vCPUs m_j.

Once the period is decided thanks to the methods proposed in this document, a value for the budget parameter is also needed. This parameter can be estimated using the monitoring tool presented in D3.4.

The one-vCPU-per-task approach. If the VP is implemented by a container, the host operating system has complete visibility of all the tasks running in the container. Therefore, a simple - yet effective - option could be to assign a SCHED_DEADLINE vCPU to each task, setting its budget to $Q_{j,i} \geq C_i$ and $P_{j,i} \leq T_i$.

This parameter assignment guarantees that each job of τ_i always receives at least the C_i time units required to complete before the next activation, which occurs with period T_i [33]. Clearly, this can only be guaranteed if the physical platform is not overloaded. For example, if vCPUs are assigned to physical

cores following a partitioned scheduling approach, the physical core in which $c_{j,i}$ is allocated must not be overloaded: this must be verified by checking that the sum of the ratios of the budgets and periods of all the vCPUs allocated to a physical core is less than or equal to one [39].

When using this approach, the number of vCPUs m_j is equal to the cardinality of set Γ_j ($|\Gamma_j|$).

When VPs are implemented by a KVM-like virtual machine, this approach is not possible. Indeed, the host operating system has no visibility for the tasks running inside the VM but only visibility about the Linux processes that implement the virtual CPUs of the virtual machine. Also, scheduling the individual tasks with the SCHED_DEADLINE policy of the guest kernel would not lead to the intended temporal behaviour since the VM is subject to the scheduling effects occurring at the host operating system level. Since the guest kernel sees the host's "real-time", every time the VM is preempted, the tasks running in the guest would be accounted for the wrong runtimes (including the time for which the VM did not run).

To overcome this issue, the *m-vCPU approach* can be used.

The m-vCPU approach. The m-vCPU approach considers a fixed number m_j of vCPUs to implement the VP v_j, allowing vCPUs to manage multiple tasks. As previously discussed, this approach is more natural for using SCHED_DEADLINE reservations for the processes implementing the vCPUs of a VM under KVM-like virtualization.

Furthermore, this approach can also be used when using containers to simplify the decision-making problem: for example, if m_j is fixed, the budget and periods of all the vCPUs can be set to the same value ($Q_{j,1} = \cdots = Q_{j,m_j} = Q_j$) and $\left(P_{j,1} = \cdots = P_{j,m_j} = P_j\right)$, and the set of parameters needed to identify the timing behavior of a VP just consist of the triplet $\left(Q_j, P_j, m_j\right)$.

In this case, suitable budget and period parameters can be achieved using methods from the real-time systems literature for the design of the parameters of reservation servers. A vast literature exists on this topic; however, since in an edge architecture these parameters need to be defined online, we refer the interested reader to works [40], [41], that provide heuristics methods for designing the reservation budgets and periods in a few milliseconds.

Running PeriodWiz inside a VP. The implementation choice for the virtual platform not only influences the assignment of the reservation parameters from tasks' parameters but also affects how PeriodWiz can be used.

If the virtual platform in which the application is running is based on a Docker-like container, the host kernel has complete visibility of the application's tasks, hence the PeriodWiz daemon can run on the host and can trace the tasks to identify their periods without issues.

If, instead, the virtual platform is based on KVM-like virtualization, then the host kernel only sees the VM's vCPU threads. Hence, PeriodWiz cannot directly trace the application's tasks to detect their periods. In this second case, there are various possibilities:

- The PeriodWiz daemon can be executed inside the VM; in this case, if the host scheduler does not affect the applications' activation pattern, the daemon can still detect periodic applications and their periods.

- The PeriodWiz daemon can be executed in the host to analyze the activation pattern of the vCPU threads.

- The applications' activation patterns can be detected before starting the applications in the virtual platform by running the application in a container or on a different node, where PeriodWiz can analyze it.

We consider some of these cases later in the evaluation.

Evaluation. We now evaluate the effects of the number N of samples using our frequency-estimation mechanism on 3 different applications: a synthetic real-time application composed of a periodic task with period $P = 40ms$, the ffplay video player reproducing a video (with its synchronized audio track) at 33 frames per second (FPS), and a non-periodic application performing some processing on data stored on the disk.

Figure 5 shows the energy $S_i(f)$ of the Fourier transform for the periodic task, with $N \in \{5,10,20\}$ samples. The figure shows how increasing the number of samples increases the energy of the frequency peaks as well, making it easier to detect them. Our tool is able to identify the program as periodic (with the correct period $T = 40ms$) when $N = 10$ or $N = 20$ samples are used; hence, the minimum delay for identifying the task as periodic is $\delta = 40ms \cdot 10 = 400ms$.

Figure 6 shows the energy $S_i(f)$ of the Fourier transform for the audio/video player, computed on $N \in \{10,20,50\}$ samples (in this case, $N = 5$ did not provide any useful information). Although this application does not exhibit a clearly periodic activation pattern (it has to display a video frame every $33.3ms$, to periodically decode and play the audio track, to read the compressed data from disk, etc...) the energy $S_i(f)$ allows to identify some peaks. However, such peaks are visible only when enough samples are used; in particular, the application is able to identify them for $N = 50$.

Finally, Figure 7 shows the energy $S_i(f)$ of the Fourier transform for a non-periodic application, computed on $N \in \{20,50,100\}$ samples. In this case, it is clearly not possible to identify any peaks in the energy function, and PeriodWiz marks the application as "not periodic".

Moreover, PeriodWiz has been tested to analyze the activation patterns of various periodic tasks, using cyclictest[20], rt-app[21] and some synthetic real-time applications, and it was always able to correctly identify such applications as periodic (with the correct period). It has also been tested with some "almost periodic" applications (such as audio/video players), and it was able to detect periodic activation patterns. However, the presence of aperiodic components in the application caused deviations from the expected period. For example, an audio/video player reproducing video at 30FpS (i.e., the expected period is 1/30 seconds, that is, 33.3 ms) was detected as periodic with a period $T = 11.1ms$, probably because audio decoding/reproduction and file parsing introduced some high-frequency components. Nevertheless, $11.1ms$ is a suitable period for an application at 30FpS (hence with a video period of $33.3ms$) since it is a sub-multiple of the video period.

[20] The Linux Foundation, "Cyclictest", Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

[21] Github, "rt-app", Available: https://github.com/scheduler-tools/rt-app

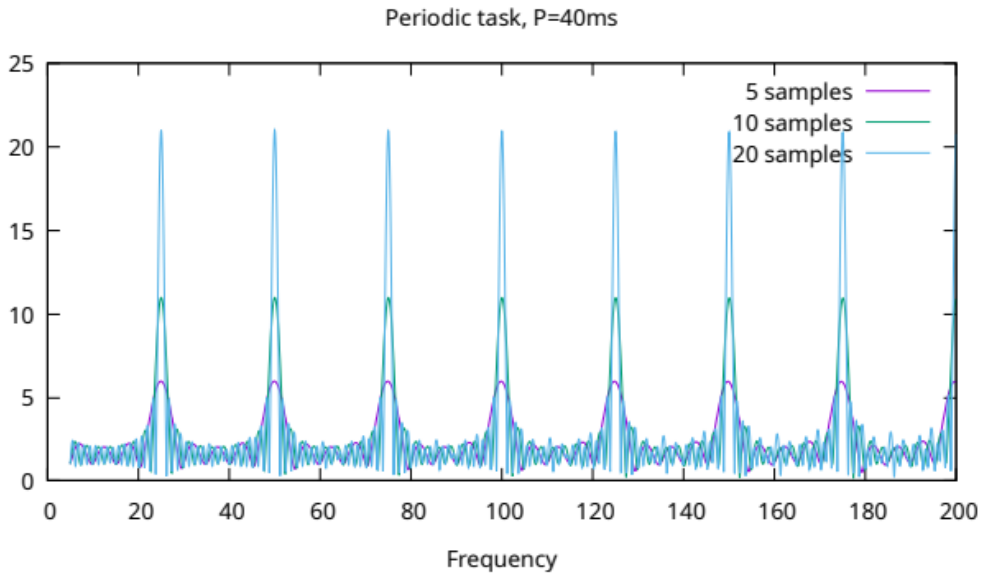


Figure 5: Energy of the Fourier transform of the activations for a periodic task with period T=40ms, for different numbers of samples.

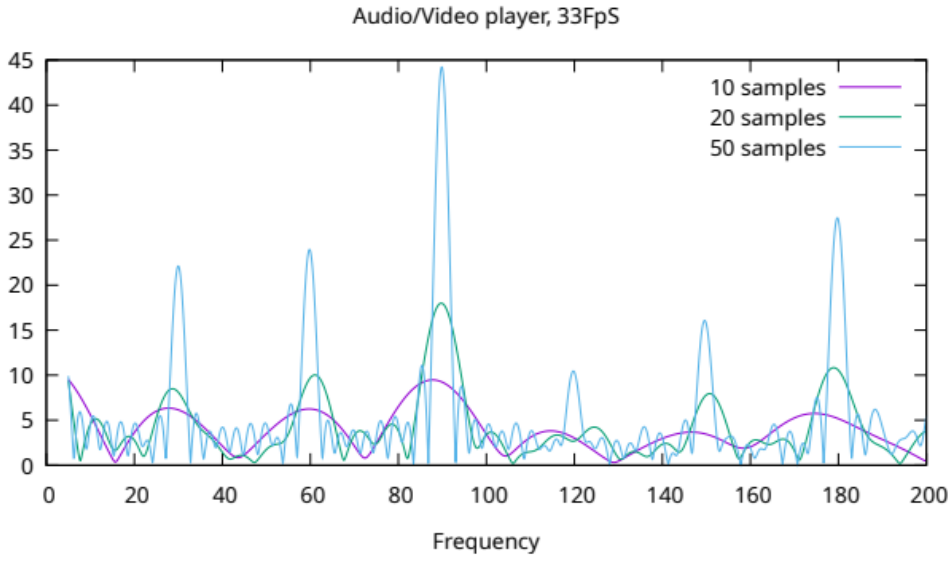


Figure 6: Energy of the Fourier transform of the activations for an audio/video player, for different numbers of samples.

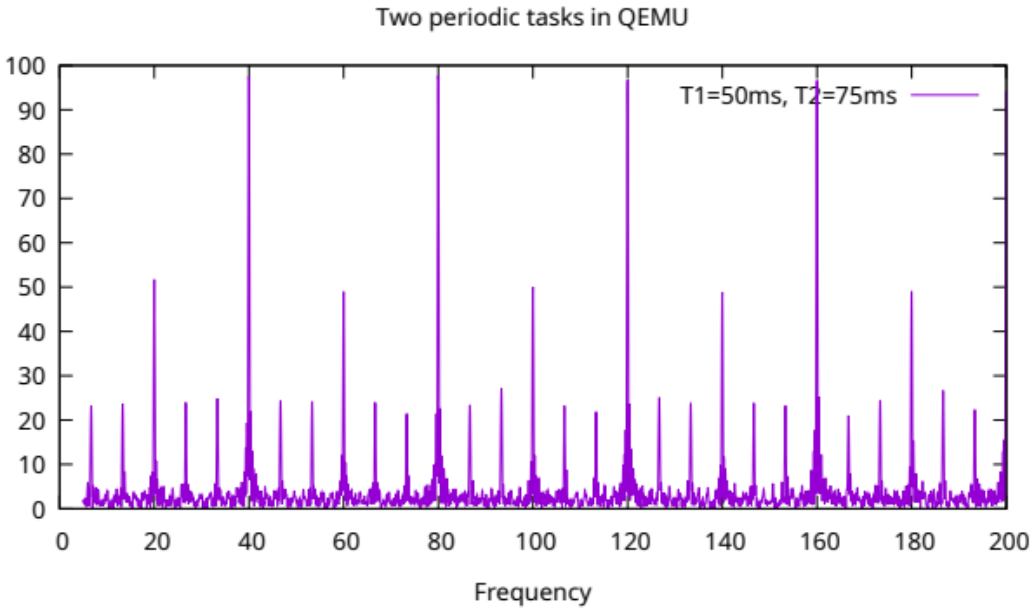


Figure 7: Energy of the Fourier transform of the activations for a non-periodic task, for different numbers of samples.

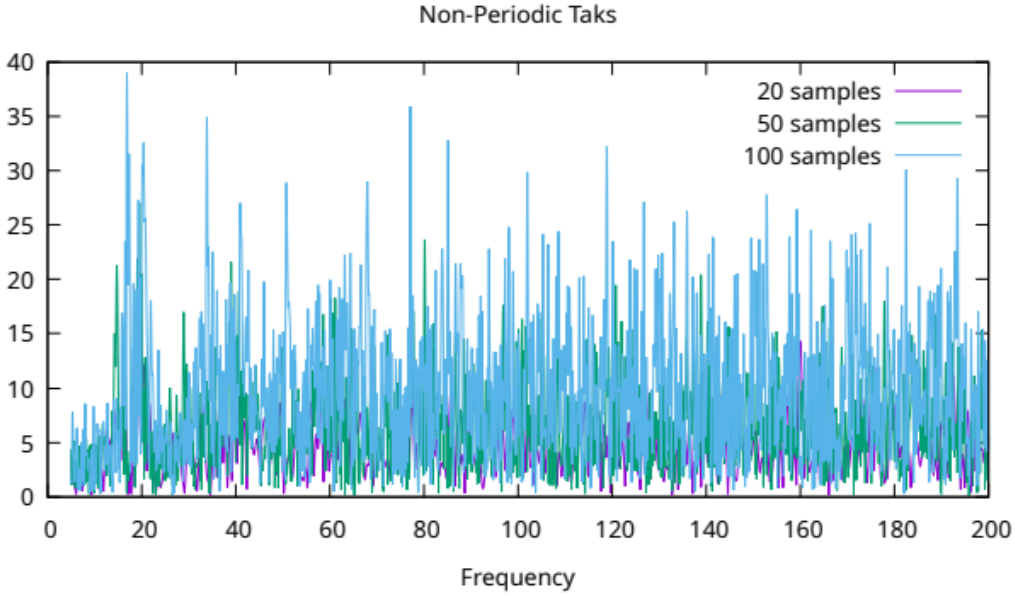


Figure 8: Energy of the Fourier transform of the activations for the QEMU vCPU thread when two real-time tasks with periods T_1=50ms and T_2=75ms run inside the VM.

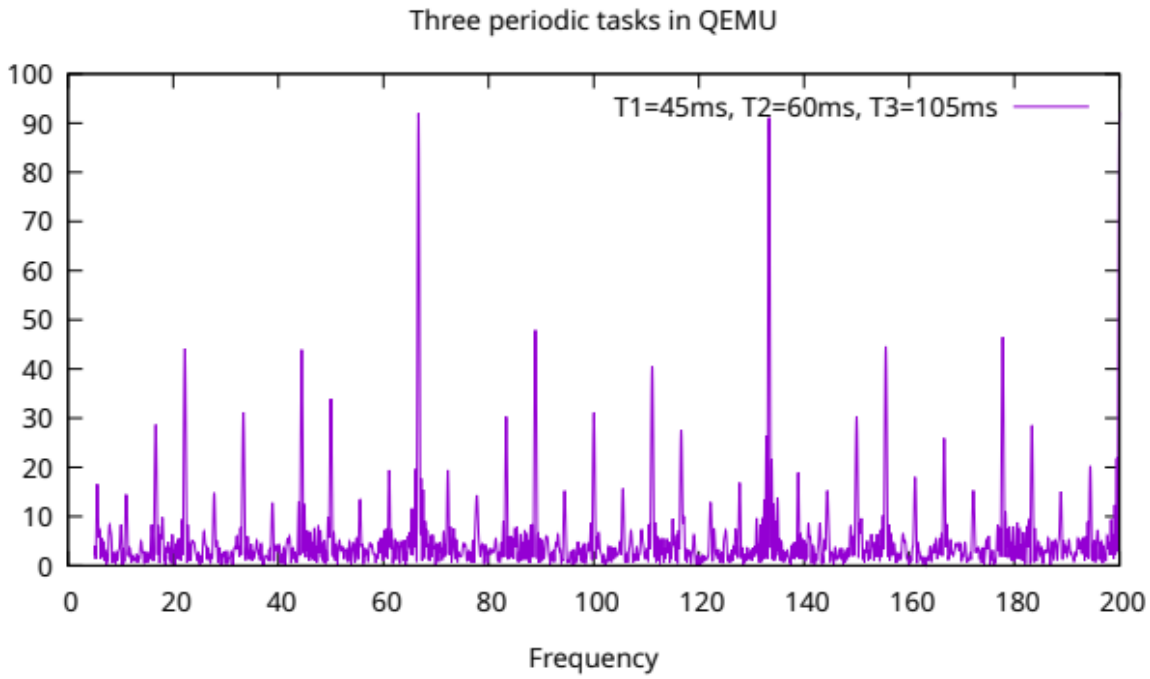


Figure 9: Energy of the Fourier transform of the activations for the QEMU vCPU thread when three real-time tasks with periods T_1=45ms, T_2=60ms, and T_3=105ms run inside the VM.

Finally, some experiments have been conducted to check how PeriodWiz performs when trying to identify tasks running in a QEMU/KVM VM. To this end, some periodic task sets have been executed inside QEMU/KVM VMs, using the PeriodWiz daemon to analyze the activation pattern of the QEMU's vCPU threads. For example, when running two periodic real-time tasks with periods $T_1 = 50ms$ and $T_2 = 75ms$ in a single-CPU VM, PeriodWiz identifies the vCPU thread as periodic with period $T = 25ms$; Figure 8 shows the energy of $S(f)$, which has a peak in $f = 40Hz$ (corresponding to $T = 1000ms/40 = 25ms$) allowing to identify the period. Similarly, Figure 9 shows the energy of the Fourier transform when three tasks with periods $T_1 = 45ms$, $T_2 = 60ms$, and $T_3 = 105ms$ run inside the VM. In this case, PeriodWiz identifies the vCPU thread as periodic with period $15ms$. More experiments revealed that PeriodWiz is generally able to identify the greatest common divisor of the periods of the tasks running in the VM; this is actually a very good choice for the vCPU's reservation period. Hence, we conclude that this approach is usable for hypervisor-based VMs, too.

4.1.2 Vertical Scaling with SCHED_DEADLINE

Figure 10 shows a reference architecture for edge systems using SCHED_DEADLINE reservations. The figure shows a distributed runtime decision-making logic with orchestration capabilities (e.g., the SCHED_DEADLINE-aware versions of Kubernetes developed in the context of Task 3.3). In the NANCY architecture, this distributed decision-making logic with orchestration capabilities (time-sensitive orchestrator, for short), can perform allocation decisions on a slice of computational resources provided to the time-sensitive domain under consideration by a higher-level orchestrator, such as the slice manager or Maestro, which is in charge of the coarse-grained allocation a subset of computing nodes to the time-sensitive domain.

In this context, the orchestrator for time-sensitive resources discussed here receives offloading requests for applications from mobile devices, which require to be allocated in a VP on the available edge nodes. Using SCHED_DEADLINE involves setting the budget and period parameters. Computing nodes report to the time-sensitive orchestrator monitoring data and receive updated values for the budgets and periods of the vCPUs implementing each VP. When an offloading request is received by the orchestrator, the periodicity of the application's tasks is estimated with PeriodWiz. As previously

discussed, PeriodWiz can be either running on the same edge node of the deployed VP or in a remote node (e.g., together with the orchestrator).

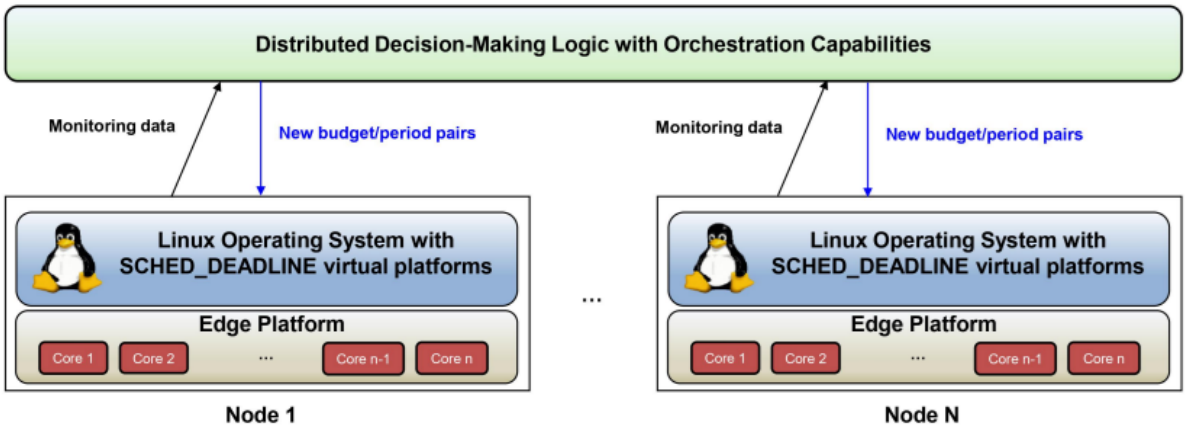


Figure 10: Schematics showing the interconnections between nodes and a distributed decision-making architecture, in which monitoring data is provided to allow vertical scaling.

Scaling. Once the period is estimated, the runtime monitoring mechanism developed in Task 3.4 of NANCY can be used to monitor the timing behavior of SCHED_DEADLINE reservations. An example is reported in the following. Two tasks are served by two reservations. Both reservations were detected to have a period of 100ms by PeriodWiz.

The two tasks are characterized by dynamic execution times: the first task initially exhibits an execution time of 5ms, then 15ms, then stabilizes to 10ms. The second task instead ranges from 10ms, to 30ms, and finally to 20ms. In this example, the period is kept constant, but variations in the period can also be detected by running PeriodWiz if needed (i.e., if another task dynamically joins the reservation). Figure 11 and Figure 12 report the results in terms of CPU bandwidth and worst-case CPU service latency of the two reservations corresponding to the two tasks.

The figures show how our monitoring mechanism is able to elastically provide more bandwidth to the reservations (and hence to the tasks) as soon as their execution time increases. Furthermore, Figure 12 shows the effects on the worst-case CPU service latency, computed as discussed at the beginning of the section. In this example, we assume all latency values are acceptable for the reservation timing constraints; when this is not the case, this information can be used by the scaling mechanism to further enhance the assignment of parameters.

Overall, the mechanisms provided in Task 3.4 and Task 4.2 provide effective ways to enable the vertical scaling of SCHED_DEADLINE reservations, achieved by dynamically changing the budget and period parameters, which directly determine the reservation of the CPU bandwidth.

These techniques can be used to adjust the resource poll of B-RAN workloads with the goal of providing sufficient resources for all functions. When this is not possible, the solution can also provide mechanisms to allow graceful degradation by reducing the amount of resources assigned (reducing the budget and/or increasing the period).

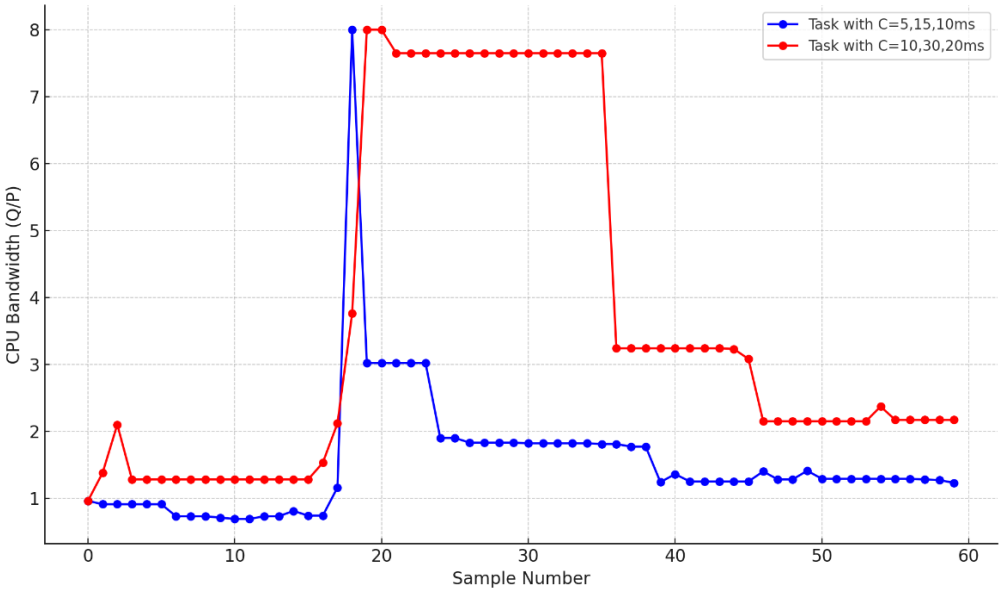


Figure 11: Elasticity in the CPU bandwidth.

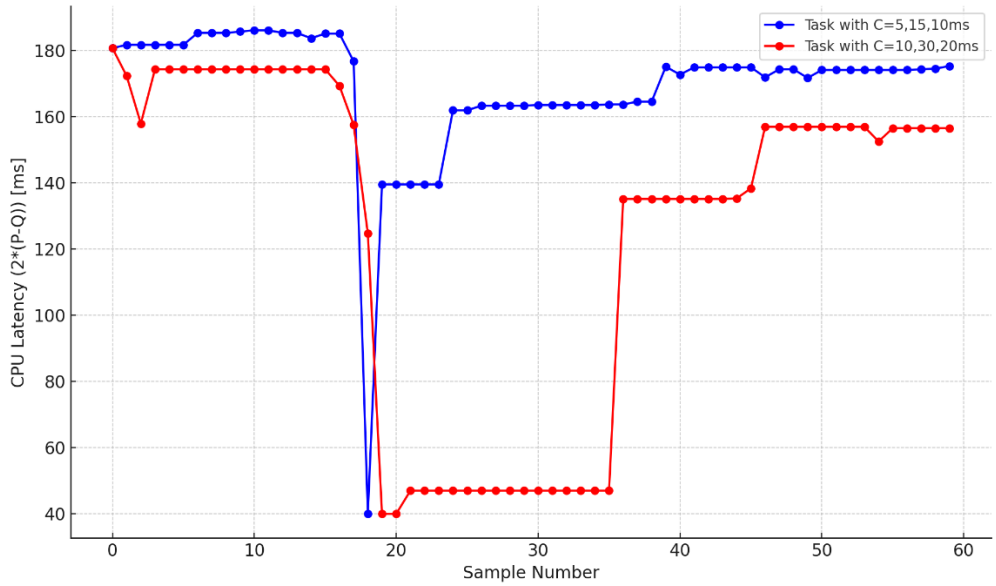


Figure 12: Worst-case CPU service latency under reservation-based scheduling.

Finally, Figure 13 shows another run of the same experiment on a higher number of samples and tracking a different metric, i.e., the reservation budget (runtime).

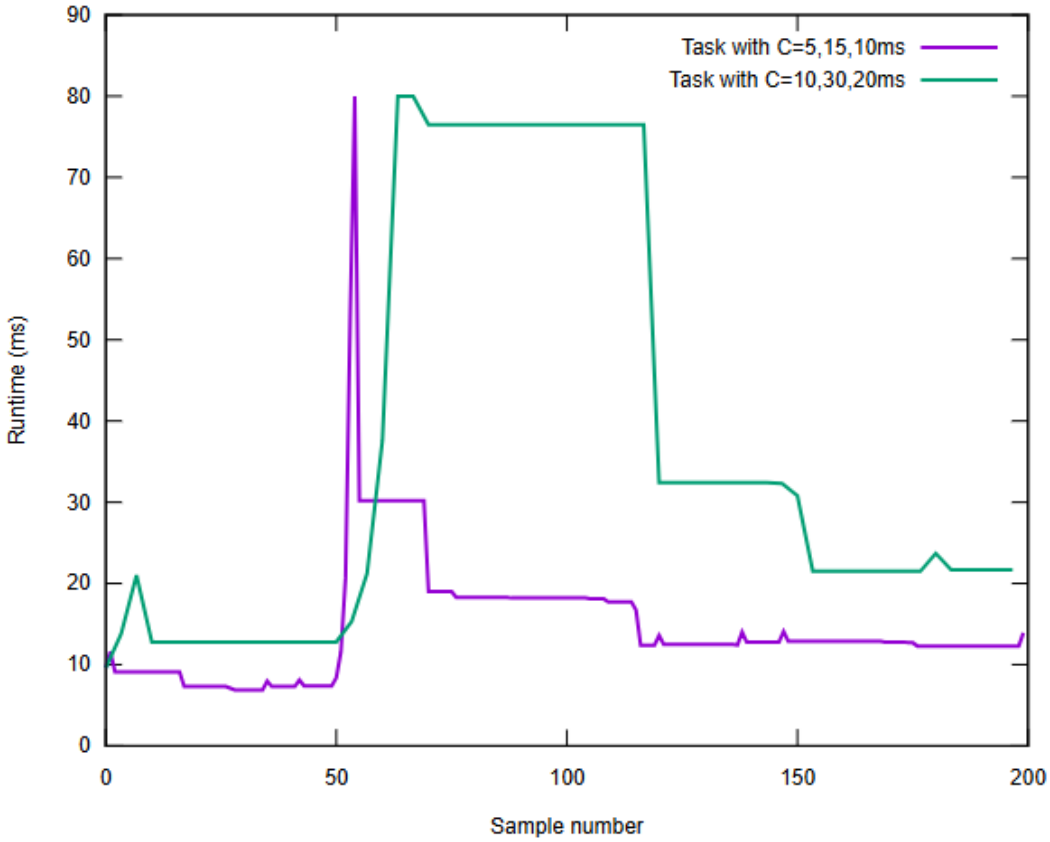


Figure 13: Reservation budget.

4.2 PHaul- a DRL-based Path Allocation for Sub6 Enhanced IAB Networks

Relying on dedicated fiber for backhauling presents a significant obstacle to the large-scale and dense deployment of outdoor small cells. In Release 16, 3GPP introduced the Integrated Access and Backhaul (IAB) technology to address this issue [42], which allows the utilization of the same spectrum in access and backhaul. IAB enables operators to start deployments with a small number of fibers connected mm-wave small cells, known as donor nodes in IAB terminology, and to extend service coverage by deploying additional IAB nodes that are wirelessly backhauled via donor nodes. These nodes can provide direct service to User Equipments (UEs), or act as parent nodes for other IAB nodes, extending the multi-hop or mesh backhaul network.

While the IAB model defined by 3GPP is spectrum agnostic, allowing operation either in the mm-wave or sub-6 GHz spectrum, the main use case for IAB networks is to provide mm-wave spectrum in the access and backhaul, complementing the Sub6 spectrum offered by the macro-cell layer. However, the performance of the mm-wave IAB backhaul segment depends heavily on the availability of line of sight (LoS) conditions in the selected deployment sites. To mitigate LoS dependence, in NANCY, we propose to complement the mm-wave backhaul segment of IAB networks with additional Sub6 backhaul links, which contribute to the capacity and robustness of the backhaul network. We refer to such networks as Sub6 enhanced IAB networks.

3GPP has defined the Backhaul Adaptation Protocol (BAP) which uses source routing to route IAB flows according to a classifier matching IAB flows into pre-provisioned backhaul paths. In this context, we have designed and evaluated the PHaul solution, a forwarding engine for Sub6 enhanced IAB networks

that accommodates different traffic engineering criteria or SLAs to optimize IAB path allocation. PHaul combines an offline path selection heuristic with an online Deep Reinforcement Learning (DRL) agent based on Proximal Policy Optimization (PPO). By leveraging a network digital twin of the IAB wireless backhaul, PHaul periodically monitors the input traffic of the backhaul network and updates flow to path mappings, performing decisions on the non-real-time domain. Although the results are focused on the IAB scenario, the proposed path forwarding engine could also be applied to other multi-hop or mesh scenarios within the scope of NANCY's User Case 2 "Advanced coverage expansion", provided that the main requirements are met: stable multi-path topology, low link variability and minimal interference. In network slicing environments, the availability of paths and the targeted SLAs for each slice would be defined by the Virtualization Platforms defined in Section 3, establishing the action space, state space and rewards to be used by the PHaul agent and within the digital twin.

In the next subsections, we present the key points of the design and implementation of the PHaul solution, together with the most significant results of its evaluation. Additional information and evaluations can be found in the referred journal paper [33].

4.1.1. PHaul Design

Figure 14 depicts the network model being considered. In the bottom left of the figure, we depict the logical architecture of a Sub6-enhanced IAB node, which, following the IAB architecture, features a Distributed Unit (DU) and Mobile Terminated (MT) function both for the Sub6 and the mm-wave bands. However, only the mm-wave DU will be used to connect UEs in the access of IAB nodes, as Sub6 access coverage is already provided by the macro layer. The donor nodes have a wired connection to the Centralized Units (CU), which in this scenario are common to all the donor nodes. The main elements of the Figure 14 are described as follows:

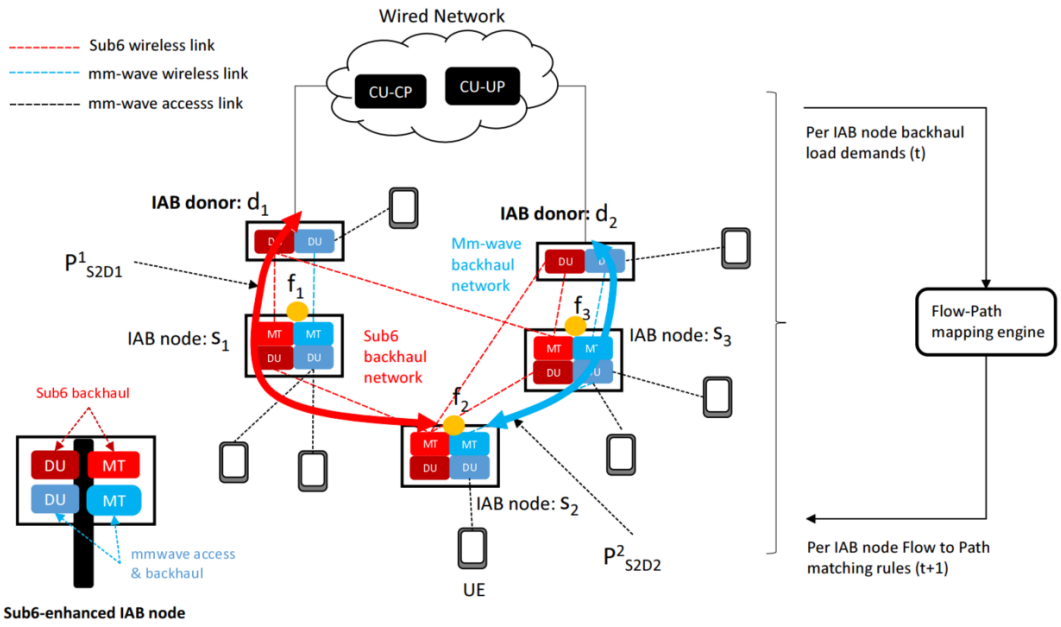


Figure 14: PHaul network model

I. **IAB donor nodes**, denoted as $d_k \in D$, featuring both a Sub6 and a mm-wave DUs, which connect UEs and other IAB nodes to the wired network, where the Centralized Unit (CU) components are located.

II. **IAB nodes**, denoted as $s_i \in S$, provide access to UEs and other IAB nodes. An IAB node may directly connect to an IAB donor or to another IAB node. Multi-path is supported by

embedding multiple MT functions into a single IAB node or by using Dual Connectivity. IAB nodes can establish more than one link to other IAB donors or IAB parent nodes. IAB nodes provide access to multiple UEs, where each UE establishes a separate PDU session. However, we consider that in the backhaul, traffic from all UEs is aggregated into a single backhaul PDU session, which we refer to as a backhaul flow.

III. **Backhaul flows**, $f_i \in F$, originate at non-donor IAB nodes. Each backhaul flow carries a time-varying traffic load, $\lambda_i(t)$, generated by all the UEs in *RRC_CONNECTED* state. Each backhaul flow is mapped to a pre-provisioned IAB backhaul path defined between an IAB and an IAB donor. For each flow, the source is fixed, i.e. s_i however any donor d_k could be used as destination, as long as a backhaul path exists. The reason is that all donor nodes provide access to the wired network where the CU resides.

IV. The **Sub6 backhaul**, shown in red in Figure 14, and the mm-wave backhaul, shown in blue in Figure 14, provide a set of backhaul paths between IAB nodes and IAB donors. Due to different propagation characteristics in the Sub6 and mm-wave bands, different paths may be available in the Sub6 and mm-wave backhaul networks. The BAP protocol in IAB uses source-based forwarding. Hence, each flow f_i is bound to use a single path, where paths are pre-provisioned. In particular, we consider that a set of $k \leq K^{max}$ paths are pre-provisioned for each flow in the Sub6 and the mm-wave backhaul networks, where $P^{s}_{ni,dk}$ indicates the n-th pre-provisioned path for flow f_i between s_i and d_k. Thus, a given backhaul flow could be routed across a total of $2K^{max}$ paths considering both networks but can only use a single path at a given time.

V. Finally, **a flow-path mapping engine** is defined as a control plane entity that periodically obtains the state of the network, e.g., through reading load counters in the IAB nodes. Based on the obtained information, the flow-path mapping engine updates the mappings between backhaul flows and backhaul paths in each IAB node. The PHaul agent resides in the flow-path mapping engine. Notice that various backhaul flows can traverse the same backhaul node and compete for the capacity of a given Sub6 or mm-wave backhaul link. In this case, we assume that backhaul nodes assign capacity to each flow traversing a backhaul link using a max-min fairness criteria.

Based on the described network model we can define the set of paths available to flow f_i as $P_i^{selected}$, with $P_i^{selected} \leq 2K^{max}$. Periodically, the flow-path mapping engine samples the traffic matrix from each backhaul flow and updates the per-flow path allocations. The goal of PHaul is to, based on the current traffic matrix $\{\lambda_i(t)\}$, assign for each flow f_i a single path $P^{opt}_{si,di} \in P_i^{selected}$, to optimize a given traffic engineering criteria J^{TE}. The execution time of the PHaul agent is the key factor to determine how often path allocations can be adapted to the varying traffic matrices. Regarding the traffic engineering criteria, we consider J^{TE} to be a varying objective function defined by the operator of the IAB network as a function of the effective data rates, $\varphi_i(t)$, allocated to each backhaul flow, where the effective data rate represents the actual rate that can be served from that flow, as compared to the overall flow demand represented by $\lambda_i(t)$. In the case of the evaluation presented later, we considered the following engineering criteria:

I. Throughput efficiency: The goal is to maximize the proportion of load that can be carried by the network. This traffic engineering criterion would be appropriate in networks where demand is expected to be heterogeneous across IAB nodes.

$$0 \leq J^{TE} = \frac{\sum_i \phi_i(t)}{\sum_i \lambda_i(t)} \leq 1 \qquad (1)$$

II. Fairness: The goal is to maximize throughput fairness across the $|F|$ flows. This traffic engineering criterion would be appropriate when demand across IAB nodes is expected to be uniform.

$$0 \leq J^{TE} = \frac{(\sum_i \phi_i(t))^2}{|F| \sum_i \phi_i(t)^2} \leq 1 \qquad (2)$$

Although not considered in the introduced evaluation, a weighted criteria combining fairness and throughput efficiency according to a weight, will allow the operator to decide on the criteria according to its business goals or Service Level Agreement (SLA).

Once the network model is defined, let us introduce the design principles of PHaul's DRL Agent. The goal of this agent is to periodically collect traffic demands for all the flows and make a path allocation decision to optimize the traffic engineering criteria. One possible implementation could consider jointly allocating all the flows |F| in each DRL action, but this would lead to an action space of size |A| = $(2K^{max})^{|F|}$, which wouldn't be feasible for practical networks. Therefore, in PHaul we considered a reduction in the action by which consists of the allocation of a path for a single flow, instead of a joint allocation for the |F| flows. The agent, which performs on a digital twin of the IAB backhaul network, observes the resulting reward of the applied action over the digital twin, and continues sampling the reduced action space until an allocation for all the flows is obtained, which can then be programmed over the real network. The rationale behind this design approach is the following:

1. The topology and link characteristics of the Sub6 enhanced IAB network are expected to be stable and can therefore be easily reproduced by a digital twin (e.g. a network simulator). If the topology changes significantly, e.g. due to link failure, then the digital twin can be correspondingly updated.
5 The traffic demands can also be considered stable for periods of several seconds and can therefore be modelled in the digital twin. This is the time budget available to perform an allocation decision.
6 Reducing the action space, i.e. allocating a path for only one flow at each action, simplifies training and leads to better convergence properties of the DRL agent.
7 The repeated application of the simplified agent over the digital twin should allow to sample the larger action space leading to well-performing allocations.

Figure 15 describes the high-level operation of PHaul. First, we distinguish the two modules that compose PHaul, namely: i) *the path computation heuristic*, and ii) *the path allocation agent*.

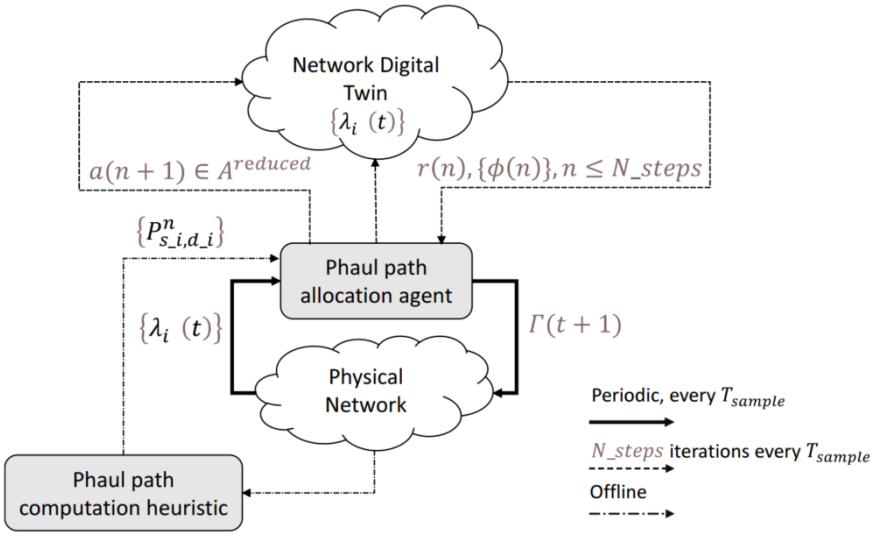


Figure 15: PHaul agent design

The path computation heuristic operates offline, and its goal is to examine the topology of the physical network to derive a set of up to K^{max} paths for each backhaul flow through both the Sub6 and the mm-

wave backhaul networks. We refer to this set of potential paths as $P_i^{selected}$. The obtained set of paths is only recomputed when the topology of the physical network changes. We considered and evaluated different heuristics, such as finding the shortest path, finding the path with the least common last hop (i.e., to avoid a bottleneck in the IAB donors) and finding the path with the least common IAB nodes (i.e., avoid bottlenecks in all the IAB links). As evaluated in (Pueyo, Camps-Mur, & Catalan-Cid), combining the latter strategy with PHaul resulted in an increased effective capacity in the network by considering the path allocations of previous flows and minimizing the number of joint links across them. Therefore, this strategy will be used as the default path computation heuristic hereinafter.

Regarding the path allocation agent, the algorithm in Figure 16 depicts the operation of the PHaul agent according to the following variables:

1. *NetDTwin* is a digital twin of the wireless backhaul network, which allows to perform flow allocations and to compute the resulting per-flow effective data rates, $\varphi_i(n)$, and the resulting rewards r(n), (see Figure 15).
8 The set of paths available to a flow f_i is defined as a vector of size $2K^{max}$, with the following components $P_i^{selected}=\{P_{i,1}^{sub6}, P_{i,1}^{mmwave}, ..., P_{i,Kmax}^{sub6}, P_{i,Kmax}^{mmwave}\}$.
9 The allocation vector Γ_i, $i \leq |F|$, where $|F\}|$ is the number of flows, and the i-th component of this vector contains the index within $P_i^{selected}$ representing the path allocated to flow f_i.

Procedure `ComputeReward()`
  $\phi_i(n) = NetDTwin.EffectiveRates()$
  Compute $\hat{thr}(n)$, $\hat{fair}(n)$
  $\hat{r}(n) = (1+\gamma)\hat{thr}(n) + (1-\gamma)\hat{fair}(n)$
  $thr_{max} = \max\{thr(n), thr_{max}\}$
  $thr_{min} = \min\{thr(n), thr_{min}\}$
  $fair_{max} = \max\{fair(n), fair_{max}\}$
  $fair_{min} = \min\{fair(n), fair_{min}\}$
Procedure `Init()`
  $n = 0$
  $NetDTwin.Init(\{\lambda_i(t)\})$
  $\Gamma_i = 0, \forall i \in \mathbb{F}$
  $thr_{max} = fair_{max} = -\infty$
  $thr_{min} = fair_{min} = \infty$
  $NetDTwin.Alloc(\Gamma(n))$
Algorithm `PHaul()`
  Input: $\{\lambda_i(t)\}$, $NetDTwin$, $|\mathbb{F}|$, γ
  Output: Γ vector with per-flow allocations
  `Init()`
  while $n < N^{steps}$ *or* $r(n) < 2$ **do**
    $\phi_i(n) = NetDTwin.EffectiveRates()$
    $state = \{\lambda_i(t), \phi_i(n), \Gamma(n)\}$
    $a = PPO.action()$
    $\Gamma(a \mod |\mathbb{F}|) = \lfloor \frac{a}{|\mathbb{F}|} \rfloor$
    $NetDTwin.Alloc(\Gamma(n))$
    `ComputeReward()`
    $n = n + 1$
  end

Figure 16: PHaul path allocation algorithm

Based on the previous variables, the action space, state space and rewards used by the PHaul agent are defined as follows:

A. Action Space: It is defined as an integer $0 \leq a \leq K^{max} |F|$, where $|F|$ is the number of flows and K^{max} is the number of paths available per flow in the Sub6 and mm-wave networks. Each action a comprehends a single flow allocation to one path in $P_i^{selected}$.
B. State space: The environment state in PHaul is modeled as a vector containing the collection of input and effective data-rates for each flow, as well as the current path allocated to each flow (line 20 in Figure 16). Note that PHaul can return to a same action, as the effect will differ depending on the allocation of the other flows. Therefore, the current allocation is part of the state space.
C. Reward: The PHaul reward is modelled after the traffic engineering criteria defined by the network operator, as defined in formulas (1) and (2). As aforementioned, a weighted combination of both formulas is also possible through g as shown in formula (3).

$$r(n) = (1+\gamma)thr(n) + (1-\gamma)fair(n) \quad (3)$$

Given that we cannot know a priory how far from 1 both $thr(n)$ and $fair(n)$ will be in a practical network, we redefine the reward as:

$$-2 \leq r(n) = (1+\gamma)thr(n) + (1-\gamma)fair(n) \geq 2 \quad (4)$$

$$\hat{J}(n) = \frac{(J(n)-J_{min})-(J_{max}-J(n))}{J_{max}-J_{min}} \quad (5)$$

Where thr(n) and fair(n) are computed according to equation (5) and are constantly updated based on the rewards obtained in each interaction with the network digital twin (line 3 in Figure 16). Note that normalizing the reward to the maximum and minimum throughput or fairness values obtained throughout the N^{steps} iterations with the network digital twin is helpful for the agent to understand if the actions being applied across the different iterations are pushing the reward in the right direction
D. Termination condition: The PHaul agent terminates after executing N steps iterations over the network digital twin, or as soon as an allocation is found that results in an optimal reward (line 18). In practice, N^{steps} is a hyper-parameter between $|F|$ and $(2K^{max})^{|F|}$, that allows to trade-off accuracy and inference time.
E. Selected DRL algorithm: In this work, we base our implementation on the Proximal Policy Optimization (PPO) [10] algorithm, which achieves state-of-the-art performance across a wide-range of challenging tasks and outperforms several other DRL algorithms [44].

4.1.2. Performance Evaluation

To evaluate PHaul we have developed a flow-level simulation model in Python, connected to OpenAI's Gym environment, which is used to implement the PHaul path allocation agent based on PPO. The developed simulator can be understood as the network digital twin component, i.e. it allows to generate random backhaul topologies and traffic matrices, to allocate a backhaul flow through a given path, and to estimate the effective capacity, obtained by a flow under a given set of flow allocations.

To model representative IAB backhaul topologies, we developed a random topology generator after the exemplary IAB network reported in [45], which covers a suburban area in Chicago, US. We generate IAB topologies in the following way. First, we start with a first layer of 3 donor nodes. Then, for each donor node we generate n IAB child nodes, with n being a uniform random variable between 1 and 3. We repeat this process for each layer of IAB nodes until the target number of nodes in the topology is reached. All IAB nodes have a wireless link to their parent node. Still, we also allow for nodes to have multiple parents using a random parameter, *edge selection prob*, that adds an additional link between an IAB node and the IAB node located next to its parent in the upper layer. This parameter allows us to control the presence of multiple paths from a given IAB node to the wired network. Notice, that

while being random, this topology generation process reflects the topology formation process in a real IAB network that would start with a fixed number of donor nodes, and then progressively grow to expand the network footprint. Following our network model in Figure 14, the same IAB nodes are available in the Sub6 and mm-wave topologies, but the exact links between nodes may differ in each topology. In particular, we set edge selection prob to 0.4 in the mm-wave topology and to 0.6 in the Sub6 topology to reflect the fact that Sub6 links have wider coverage. Following this approach, we consider scenarios with a total number of IAB nodes (including donor nodes) varying from 20 to 60, which result in mean hop counts of 3.08 for the case of 20 nodes and 5.12 for the case of 60 nodes.

We model backhaul links as interference-free with capacities that are stable during the execution of the agent. In the case of the mm-wave backhaul, we assume a 30 GHz deployment with an 800 MHz carrier bandwidth, resulting in backhaul downlink per-link capacities in suburban environments that we select randomly between 800 Mbps and 1 Gbps. For the case of Sub6 links we consider an interference-free 80 MHz link with capacities that we select randomly between 200 and 300 Mbps. Regarding the input traffic matrix, we model the load offered by each backhaul flow in the following way. First, we use a random parameter, *node_active_probability*, to determine if there are active UEs in that node. In case there are active UEs, we model the resulting backhaul traffic as a uniform random variable between λ_{min} and λ_{max}, where we consider two scenarios with a growing flow size in Mbps, namely: i) λ_{min} =250, λ_{max}=500, and ii) λ_{min} =500, λ_{max}=750.

To achieve statistically significant results, every time we report a performance figure for a given network configuration, we consider at least 10 random topologies for that network configuration, and for each of those topologies, we average the results of 250 randomized input traffic matrixes. Unless otherwise stated, the PHaul path allocation agent is trained for each specific topology considered. The metrics reported in this section correspond to average values that are depicted with their corresponding 95% confidence interval, which is however too small to be clearly seen in the figures.

PHaul is compared to three alternative path allocation agents, namely: i*) brute force*, ii) *subset-sum*, and iii) *random*. Brute force is optimal in terms of the objective function since it explores all available $(2K^{max})^{|F|}$ flow allocations, but it leads to large execution times, and it can only be used for network configurations with a limited number of flows and paths. The subset-sum path allocation agent is a greedy heuristic based on the subset-sum problem. It orders the backhaul flows in decreasing order of their traffic demand and allocates each flow sequentially trying to maximize the objective function (up to $2K^{max}|F|$ steps). Finally, the random allocation simply selects for each flow one of the available paths through the mm-wave and Sub6 backhaul networks using a uniform random variable ($|F|$ steps).

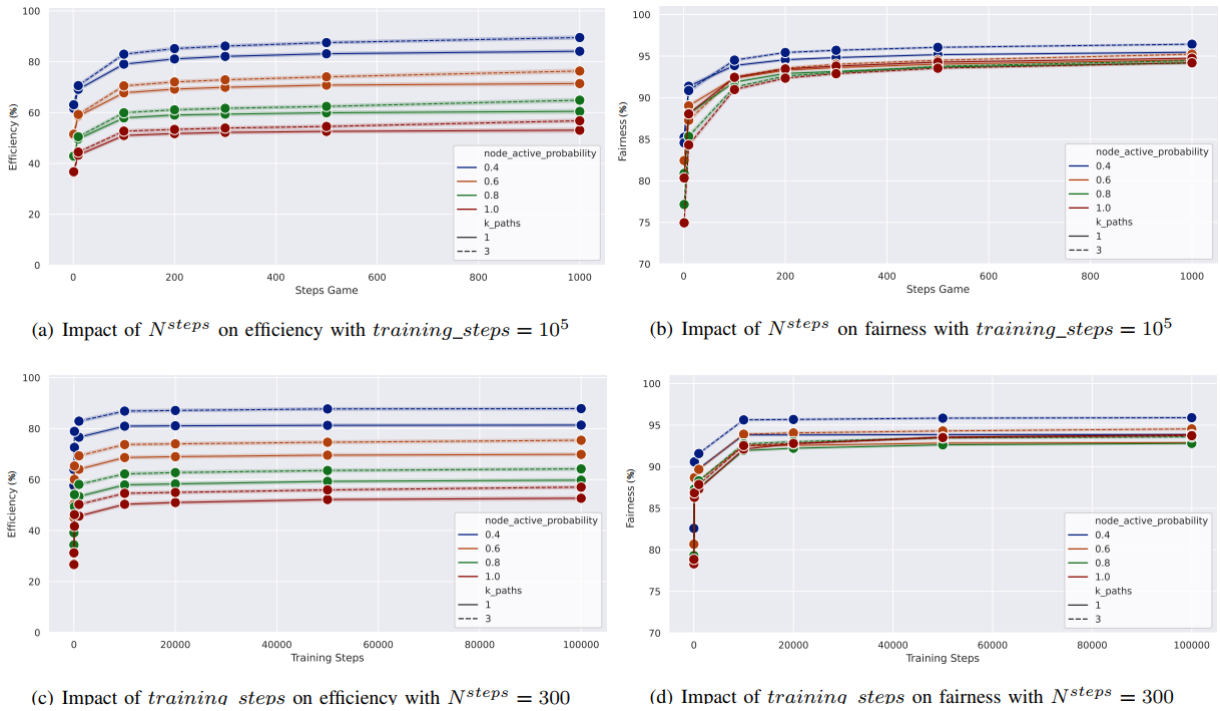
We have released as open source our implementation of PHaul, the implementation of the IAB network digital twin, and all the supporting evaluation environments in [46].

Training evaluation

We evaluate in this section the training phase of the PHaul path allocation agent. We want to understand how PHaul's performance depends on our training hyper-parameters, namely N^{steps}, and the parameter *training_steps*, which determines the overall number of steps considered in the training phase of PPO. As performance metrics, we look separately at efficiency obtained with $\gamma = 1$ in, and then at fairness, obtained with $\gamma = -1$.

Figure 17 depicts in the upper row the impact of N^{steps} while fixing *training_steps* to 10^5, and in the lower row, the impact of *training_steps* while fixing N^{steps} to 300. In these experiments, we consider an IAB network size of 50 nodes and vary the *node_active_probability* parameter between 0.4 and 1,

while considering a random backhaul flow rate between 500 and 750 Mbps. We depict experiments for K^{max}=1 and K^{max}=3 paths in the Sub6 and mm-wave networks.



(a) Impact of N^{steps} on efficiency with $training\_steps = 10^5$

(b) Impact of N^{steps} on fairness with $training\_steps = 10^5$

(c) Impact of $training\_steps$ on efficiency with $N^{steps} = 300$

(d) Impact of $training\_steps$ on fairness with $N^{steps} = 300$

Figure 17: Training evaluation

We can see how, both for efficiency and fairness, performance is low when N^{steps} and *training_steps* are small, and smoothly increases when these parameters grow. Notice though that N^{steps} will have an impact on the execution time of a trained agent, while *training_steps* only affects the overall training time, which is not a critical parameter. Looking at the knee exhibited by the training curves, we can see that this is independent of the *node_active_probability* parameter, which means that regardless of the level of activity in the network the PHaul agent training performance is maintained. The number of paths K^{max} does not impact either the position of the knee in the training curves. Based on these results the training of PHaul agent can be simplified by setting a fixed value of N^{steps} and *training_steps*, regardless of the number of paths or the number of active flows, which relaxes the requirements to train the PHaul network digital twin in real networks. Hereafter we consider N^{steps}= 300 and *training_steps*=20000.

We observe on the left part of Figure 17 how efficiency decreases when increasing *node_active_probability*. This is expected because a higher *node_active_probability* means more load being injected into the wireless backhaul, which is saturated in all cases. The impact of *node_active_probability* is however not so clear when looking at fairness (right part of Figure 17). The reason is that regardless of the number of backhaul flows active in the network, PHaul is able to allocate the bottleneck bandwidth across these flows in a fair way when considering this objective. Looking at the impact of varying the number of paths K^{max}, as expected, we observe that considering more paths leads to higher network efficiency as flows can be better balanced through the network. Notice though, that the potential gain achieved by increasing K^{max} depends on the path diversity available in the IAB topology, which is limited in our scenarios that mostly consist of tree-like topologies with limited multi-path opportunities.

Inference time

The main goal of PHaul is to periodically read the traffic matrix from the physical network to then update the mapping between backhaul flows and pre-provisioned backhaul paths. The frequency of these updates is thus limited by the execution time of the path allocation agent. Figure 18 depicts the average execution time of the PHaul, subset-sum and random agents, when increasing the IAB network between 20 and 60 nodes.

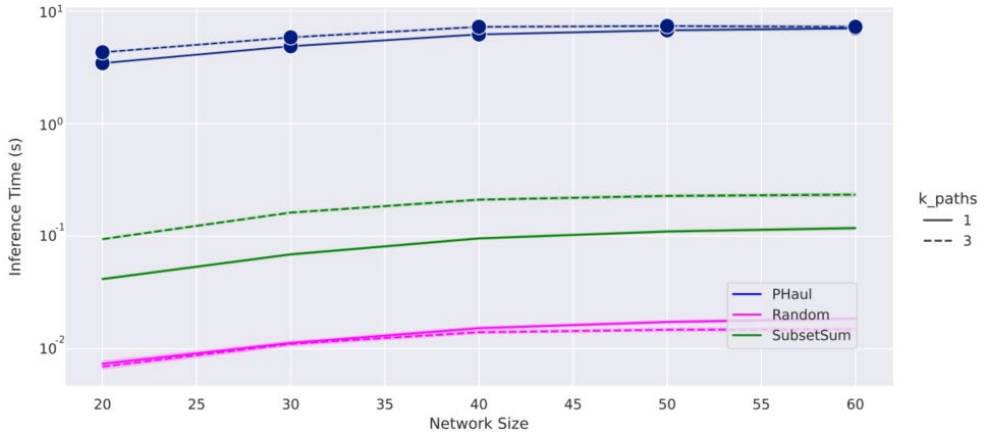


Figure 18: Inference time evaluation (Platform: Intel Xeon E5-2618L v4 CPU)

As expected, the execution times of PHaul are larger than those of subset-sum and random. However, PHaul keeps an execution time below 10 seconds which slightly increases with the network size. A 10-second interval to reconfigure the forwarding in the backhaul is a reasonable value to observe significant changes in the traffic matrix (i.e., UEs transitioning from idle to connected status, or vice versa). It is also relevant to observe how the execution time of PHaul is fairly independent of the network size. The reason is that the execution time of PHaul is dominated by the N^{steps} parameter, which defines the number of interactions with the network digital twin. It is thus possible to reduce execution time in PHaul by reducing N^{steps}, at the cost of losing accuracy in terms of the objective function, as depicted in Figure 17. Regarding the number of paths K^{max}, we can see that they only have a marginal impact on the execution time of PHaul. The reason is that increasing it translates into an increase in the size of the action space, which may impact convergence time in the training phase, but it results in a minor impact with respect to the time required to decide what action to choose in the inference phase. This is not the same for subset-sum, which is clearly affected by the number of paths, as it needs to evaluate $2K^{max}$ candidate path allocations for each flow. Finally, the execution time of the random agent is negligible, as it only involves computing a random number.

Comparison against competing heuristics

Figure 19 depicts a comparison with the brute force allocation, which illustrates how far PHaul performance is from the optimal allocation. Due to the high computational requirements of the brute force agent, we are only able to carry out this benchmark with a limited network size of 20 IAB nodes and with $K^{max} = 1$. To have a meaningful comparison, we need to ensure that the network is saturated, for which we configure *node_active_probability*=1 and use flow load between 500 and 750 Mbps. We can see in Figure 19 that both the efficiency and fairness achieved by PHaul lie very close to the brute force agent, which validates the performance of PHaul in the considered scenario. The performance gap between PHaul and brute force is however expected to increase if larger topologies are considered.

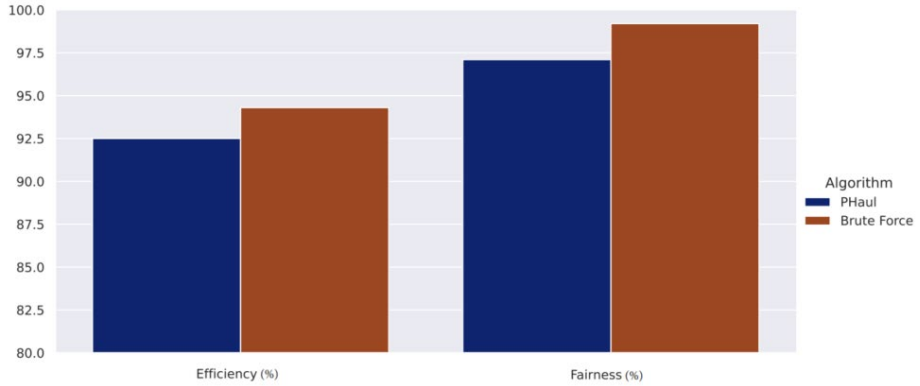
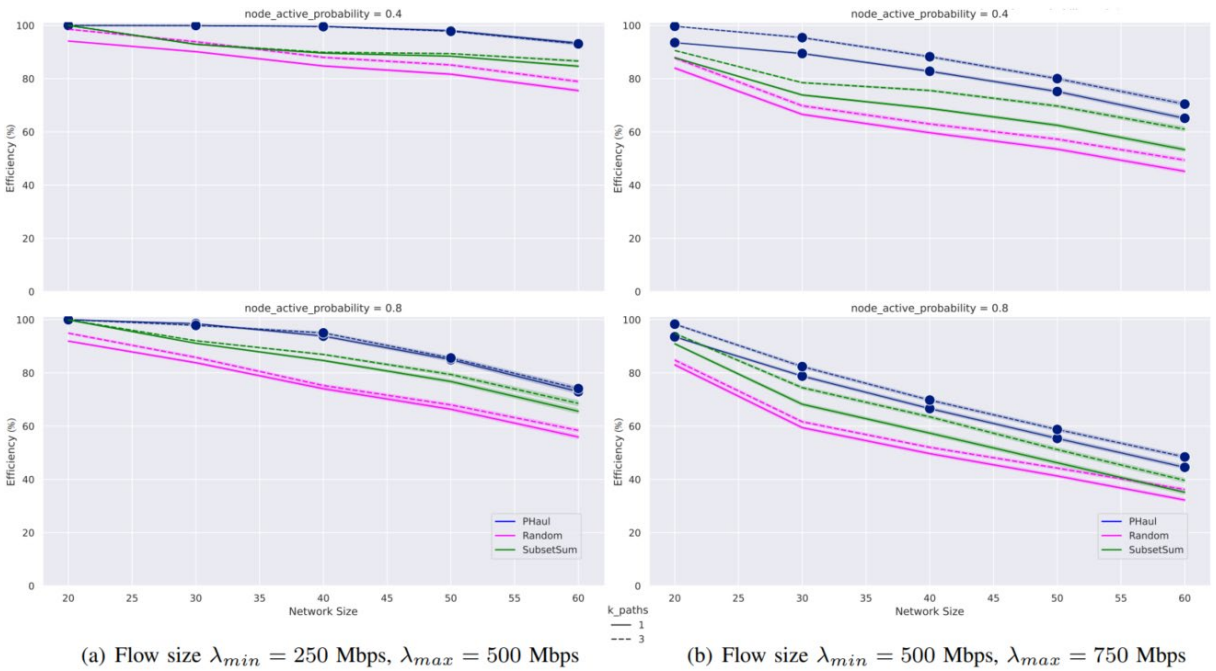


Figure 19: Comparison with Brute Force

Next, we evaluate the performance of the PHaul, the subset-sum and the random path allocation agents, when increasing the size of the IAB network from 20 to 60 nodes, including 3 IAB donor nodes. We do not consider the brute force agent in this section because of its excessive computational time.

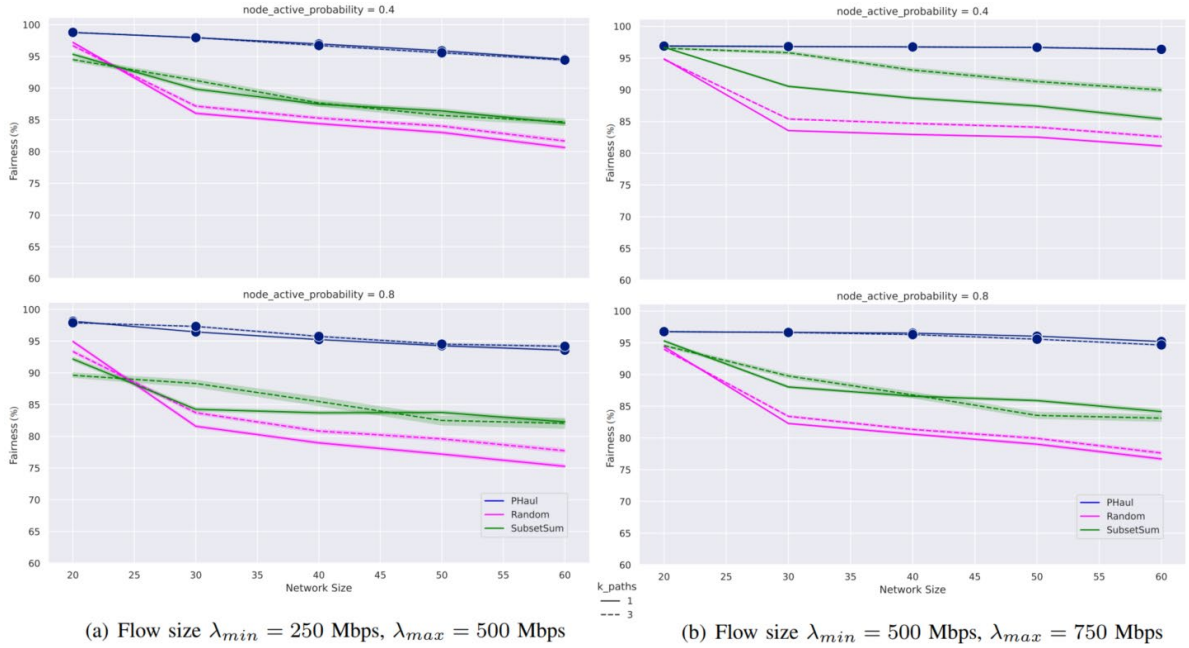
Figure 20 depicts the results for the efficiency objective under different network and load conditions. We can see how for all network configurations the PHaul agent outperforms subset-sum, which in turn outperforms the random agent. A maximum gain of 17% is observed for PHaul when compared to subset-sum, and of 36% when compared to random. All agents benefit from considering a larger number of paths, but this gain is more evident when the flow data rates are higher. Note that subset-sum is a well- bin-packing heuristic that sorts backhaul flows in decreasing order of size and greedily starts allocating them one at a time. The reason why PHaul is able to outperform subset-sum is that in the training process PHaul is able to learn a representation of the topology of the IAB network, which it can then correlate with a given traffic matrix distribution to derive non-trivial allocations that result in good performance. For all the agents, efficiency decreases as network size increases, and the decrease is higher for higher flow data rates. The reason for this behavior is that introducing new IAB nodes results in a higher offered load, regulated by the parameter OBJnode active probabilityOBJ. This effect dominates over any increase in cross-section bandwidth that may result from the additional backhaul links contributed by the new IAB nodes.



(a) Flow size $\lambda_{min} = 250$ Mbps, $\lambda_{max} = 500$ Mbps (b) Flow size $\lambda_{min} = 500$ Mbps, $\lambda_{max} = 750$ Mbps

Figure 20: Comparison with other heuristics – Efficiency

Figure 21 depicts the results in terms of fairness. Unlike efficiency, fairness exhibits a rather flat behavior when the size of the IAB network grows. In our network model, each IAB node allocates per-flow capacities in a bottleneck link using a water-filling algorithm. Therefore, the means that the different agents use to improve fairness is to select the paths for each flow such that all flows in the network achieve a similar effective capacity. We can see how PHaul is the best agent in achieving a fair allocation, resulting in a close to perfect fairness for all considered network settings, and for both 1 and 3 available paths. A maximum gain of 13% is observed in terms of fairness for PHaul when compared to subset-sum, and of 20% when compared to random. The reason for the good performance of PHaul, is that in the training phase PHaul is able to learn correlations between a given traffic matrix and the set of available paths that result in a good fairness metric.



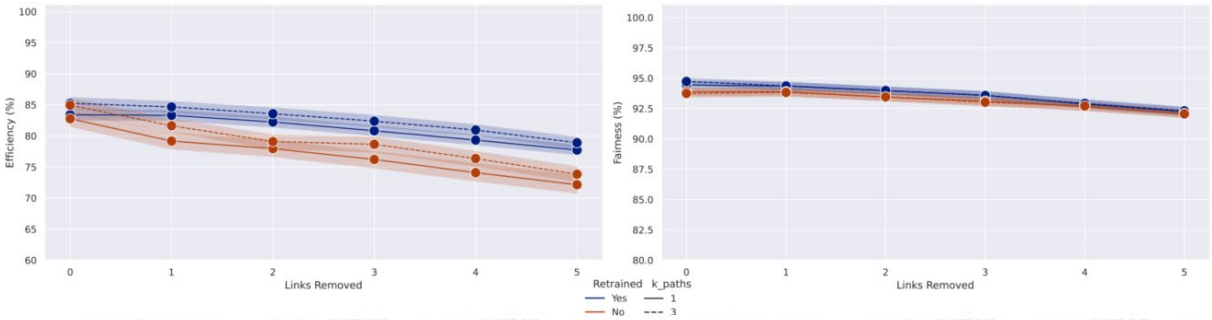
(a) Flow size $\lambda_{min} = 250$ Mbps, $\lambda_{max} = 500$ Mbps (b) Flow size $\lambda_{min} = 500$ Mbps, $\lambda_{max} = 750$ Mbps

Figure 21: Comparison with other heuristics – Fairness

Broken links

Given the nature of the IAB wireless backhaul, transitory broken links can occur due to link blockage at mm-wave frequencies, or due to unplanned interference at Sub6 frequencies. The goal of this section is to evaluate how resilient is PHaul to these events since it is not realistic to assume that PHaul can be retrained every time a link breaks. The resulting topology upon a link break will differ from the topology that PHaul has been trained on, and hence a performance degradation can be expected. The goal of this section is to quantify this degradation.

Figure 22 depicts the results of an experiment where we consider a network of 40 IAB nodes, with a flow load between 500 and 750 Mbps and *node_active_probability*=0.4. We increase the number of simultaneous broken links from 1 to 5, which we consider representative of this network size. For each point in the x-axis we consider 10 different topologies and 250 random samples with different traffic matrixes. In each sample, we randomly remove x links from the network but ensure that end-to-end connectivity for all backhaul flows remains possible. Figure 22 compares the performance in terms of efficiency and fairness, considering the ideal performance where PHaul is retrained every time that a link is removed from the topology (shown in blue), versus the performance to be expected in practice when PHaul has only been trained for the full topology but continues to perform inferences when links are removed (shown in brown).



(a) Efficiency ($\gamma = 1$) for $K^{max} = 1$ and $K^{max} = 3$ (b) Fairness ($\gamma = -1$) for $K^{max} = 1$ and $K^{max} = 3$

Figure 22: Broken links evaluation

Based on the results, the overall efficiency reduces as the number of broken links increases, because the network has less capacity, but the efficiency loss because of having PHaul operate over a network with broken links is only around 5%. In the case of fairness, we observe that the loss in fairness is even smaller, being around 2%. We note that the tree-like structure of IAB backhaul networks tends to result in backup paths that are like the original ones, which helps explain the good performance of PHaul observed in this experiment.

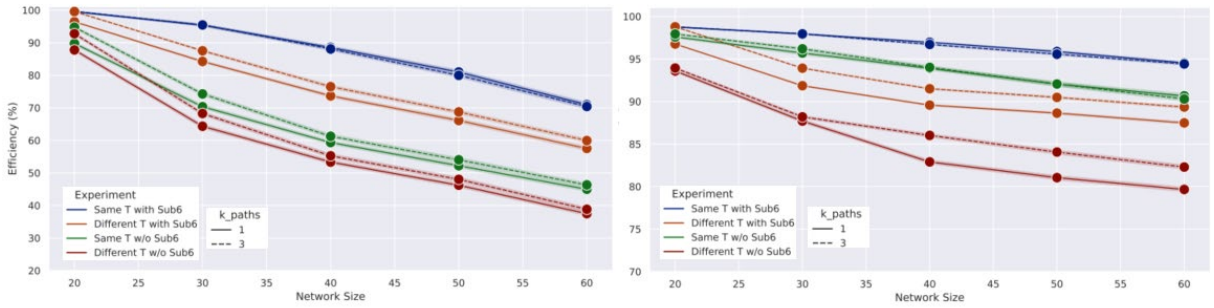
Untrained topologies and impact of sub6

Having seen in the previous section that PHaul is reliable to small changes in the topology, in this section we deepen our evaluation in two directions. First, we evaluate the performance of PHaul on topologies that differ significantly from the topology that PHaul has been trained on. Second, we evaluate the performance of PHaul with and without Sub6 connectivity, to quantify the gains that adding Sub6 spectrum provides on efficiency and fairness metrics.

To evaluate PHaul with untrained topologies, we consider again 10 topologies for each considered network size, but we trained PHaul with only one of the 10 topologies. All topologies share the same number of IAB nodes and the same number of donor nodes, which is what makes PHaul applicable across them. Topologies though, differ in the number of layers and in the links connecting IAB nodes. To evaluate the gains provided by Sub6 spectrum, we run each topology under two configurations: i)

a first configuration where both mm-wave and Sub6 spectrum are available, and ii) a second configuration where only mm-wave links are available.

Figure 23 depicts for a growing network size the performance in terms of efficiency and fairness. For this experiment, we assume a flow load between 500 and 750 Mbps and *node_active_probability*=0.4. We consider the following four configurations: i) Sub6 enhanced IAB where PHaul is trained specifically for each topology (blue lines), ii) Sub6 enhanced IAB where PHaul is only trained for one topology (brown lines), iii) mm-wave only IAB where PHaul is trained for each topology (green lines), and iv) mm-wave only IAB where PHaul is only trained for one topology (red lines).



(a) Efficiency ($\gamma = 1$) for $K^{max} = 1$ and $K^{max} = 3$ (b) Fairness ($\gamma = -1$) for $K^{max} = 1$ and $K^{max} = 3$

Figure 23: Untrained topologies and sub6 evaluation

Looking at the impact on efficiency, we can see how using untrained topologies results in a slight degradation of about 9% in efficiency. Removing Sub6 spectrum, but retraining PHaul every time, results in an efficiency loss of around 25%, which justifies the gains provided by Sub6 spectrum, since, using PHaul, a Sub6 enhanced IAB network with untrained topologies is still more efficient that a perfectly trained mm-wave only IAB network. Finally, removing Sub6 spectrum and using untrained topologies results in an overall degradation slightly above 30%. In the case of fairness, we can see that the trend is maintained, but interestingly, removing Sub6 spectrum results in a degradation of around 4%, whereas using untrained topologies result in a higher degradation of around 7%. The reason is that when removing Sub6 spectrum the overall network capacity reduces, but PHaul is still able to allocate the available capacity across flows in a fair way. Fairness is however impacted when PHaul runs over untrained topologies, although the impact is small. Finally, removing Sub6 spectrum and using untrained topologies results in the worst case in a degradation of around 15%.

These results quantify the benefits of adding Sub6 spectrum to the IAB backhaul, as proposed in NANCY, and validate that PHaul has a graceful degradation when used over topologies that differ significantly from the topologies that have been used for training, which validate the application of PHaul in practical networks.

4.3 Elastic Resource Scaling

4.3.1 Scaling for Disruption-Free Service

Scaling within a network slice while maintaining disruption-free service for NANCY B-RAN services is critical to ensuring that various key services, such as localization, data analytics, and many others, maintain seamless performance even as conditions like network load, number of requests, service demands, etc. fluctuate. The decision to adjust a slice must also ensure that when resources are shared among multiple services, key performance metrics, such as latency or throughput, as listed in Section 3.3, are not compromised, and ongoing services are not interrupted. For example, in an URLLC slice, user location data is critical for decision-making processes for autonomous vehicles. If the system cannot accurately locate a vehicle in real-time, it could lead to catastrophic outcomes, such as collisions

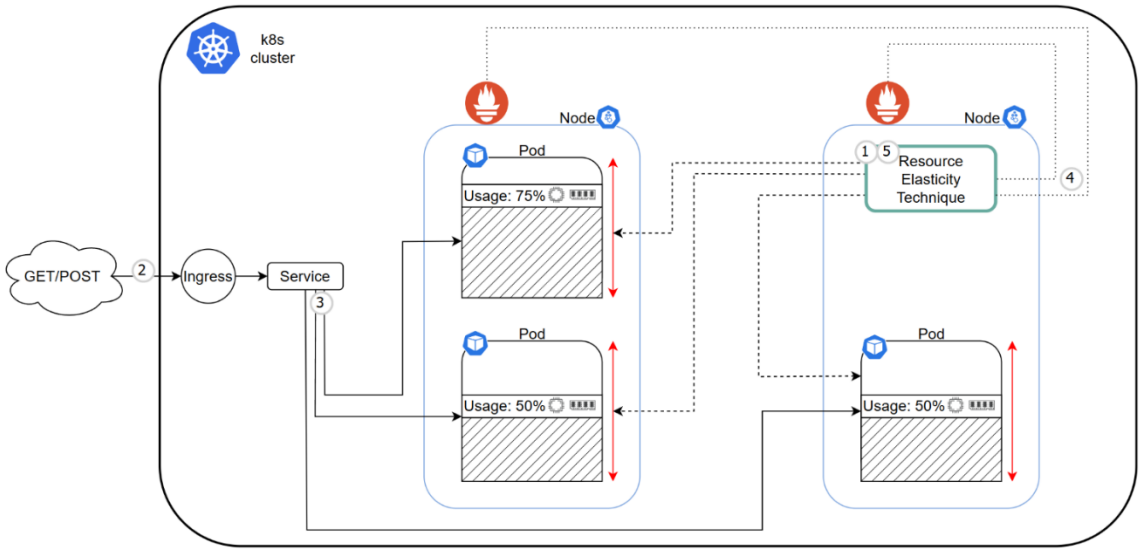
or failure to navigate safely in dynamic traffic conditions. Therefore, scaling within a network slice (vertical scaling) is essential for ensuring that resource allocation can be dynamically adjusted in real-time without affecting the reliability of these high-priority services.

To achieve elastic resource adjustments within a designed slice for multiple services, **Deep Reinforcement Learning (DRL)** has been identified as the most promising approach for real-time decision-making in dynamic network environments [13]. As mentioned in Section 2.2,by formulating the scaling problem as a **Markov Decision Process (MDP)**, DRL enables flexible, dynamic resource management by modeling the environment with states (e.g., current resource usage), actions (e.g., scaling up or down), and rewards (e.g., improved performance). A **model-free, online learning** method allows agents to learn optimal policies through trial and error, continuously updating their decisions based on real-time feedback from the network. This ensures that as conditions such as network load fluctuate, the DRL agent can dynamically adjust resources without requiring prior knowledge of transition dynamics. This adaptive approach makes DRL highly effective for maintaining optimal performance in complex 5G networks while managing multiple services within the slice.

4.3.2 Intra-slice Resource Elasticity in NANCY

The intra-slice resource elasticity in NANCY's system involves the dynamic management of resources within a Kubernetes-based system. The system is designed to ensure efficient and elastic resource allocation by handling fluctuating traffic loads, e.g., requests to perform localization service, which is developed in D3.2 as one of the common network functionalities. More specifically, the system operates by hosting containers (pods) on virtual machines (VMs), and managing the execution of these pods as they handle service requests. These service requests utilize HTTP as the protocol and are further routed through the Ingress controller, which directs them to the appropriate services. These services then distribute the requests to the relevant pods based on internal routing and load-balancing mechanisms. By continuously adjusting resources based on demand, the system ensures seamless performance under varying conditions, optimizing both resource utilization and service reliability. Moreover, this is shown in Figure 24, which provides a high-level overview of the system used for implementing elasticity, along with the five steps required to set up and dynamically allocate resources, which can be seen at the bottom of the figure.

The core of this elastic system lies in the resource elasticity technique, which continuously monitors resource usage and adjusts resource allocation dynamically to meet service demands. This guarantees that the system can handle fluctuations in traffic load by scaling resources as needed without causing service disruption. To support real-time monitoring, Prometheus is configured to collect key performance metrics at one-second intervals, such as the limitation and utilization of the CPU. This high-frequency monitoring enables the resource elasticity mechanism to respond swiftly to changes in demand, ensuring optimal performance and resource utilization across the slice.



1	The Kubernetes cluster is deployed, initial resources are allocated to its respective services
2	The client's HTTP GET/POST requests are routed through the Ingress Controller to the services for external access
3	The Ingress Controller forwards the request to the appropriate Kubernetes service, which is routed to the available pod
4	Prometheus is used to continuously monitor key performance metrics in real-time
5	Based on the monitored metrics, the resource elasticity technique dynamically enforces the service's resources

Figure 24: Overview of elastic scaling with Kubernetes-based system.

In Figure 24, Steps **1 to 5** sum up the resource elasticity process mentioned above. More specifically, in order to enable seamless resource scaling, which is shown in **Step 5,** the **InPlacePodVerticalScaling** feature gate [22] is activated, allowing for dynamic resource adjustments without requiring container restarts. This step is crucial as it ensures continuous operation and minimizes disruption when scaling resources up or down in response to fluctuating demands. Moreover, real-time resource management is facilitated through communication with the Kubernetes cluster using the **Python Kubernetes API** [23], enabling precise control over resource allocation. Commands such as "patching" are used to modify resources, allowing for the addition or removal of CPU and memory allocations as needed. This ensures efficient resource management while maintaining uninterrupted service continuity.

The code below demonstrates how the **Python Kubernetes API** is used to achieve real-time resource adjustments, further highlighting the implementation of Step 5 for maintaining operational efficiency and flexibility within the system:

```
1.  def patch_pod(pod_name, cpu_request, cpu_limit, memory_request,
2.            memory_limit, container_name, namespace="default"):
3.      config.load_incluster_config()
4.      api_instance = client.CoreV1Api()
5.      patch = {
6.          "spec": {
7.              "containers": [
8.                  {
9.                      "name": container_name,
10.                     "resources": {
```

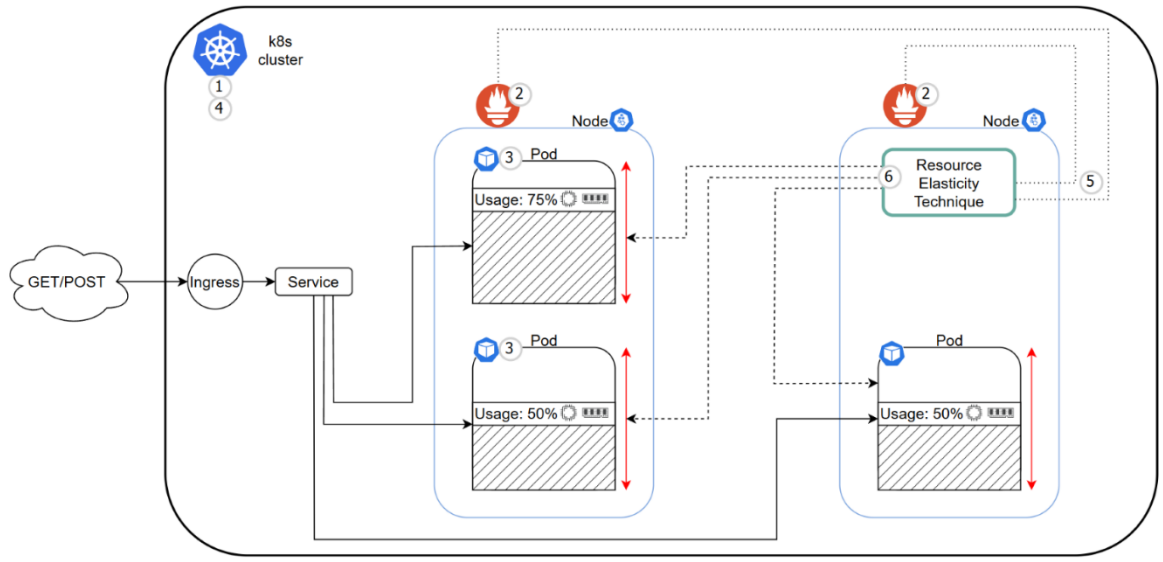
[22] "InPlacePodVerticalScaling," [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/resize-container-resources

[23] "Python Kubernetes API," [Online]. Available: https://github.com/kubernetes-client/python

```
                "requests": {"cpu": cpu_request,
                             "memory": memory_request},
                "limits": {"cpu": cpu_limit,
                           "memory": memory_limit}
            }
          }
        ]
    }
}
try:
    api_instance.patch_namespaced_pod(
      name=pod_name,
      namespace=namespace,
      body=patch,
    )
except Exception as e:
    print(f"Error: {e}")
```

4.3.2.1 Integrating Resource Elasticity with the Slice Manager API

To enable MADRL-based resource elasticity through the Slice Manager API, telemetry must first be configured as a resource within the system for monitoring purposes, as shown in Figure 25. Telemetry continuously tracks the overall health and performance of the cluster and its services. Once deployed, specific Prometheus instance details are registered with the telemetry resource, allowing it to gather real-time metrics from these instances. This provides real-time access to performance data, which can be used to dynamically manage resources and optimize the cluster's operation. In the following steps, a new slice of computation resources, referred to as compute chunks is created to handle specific workloads. Based on the telemetry data, the resources of existing chunks can be adjusted to meet demand.



1	POST /telemetry - register a new telemetry resource for monitoring the cluster
2	POST /telemetry/{telemetry_server_id}/target - register the Prometheus instances to the telemetry
3	POST /compute_chunk - creates new compute chunks
4	POST /slice3 - creates new slice of the compute chunks and services
5	Prometheus is used to continuously monitor key performance metrics in real-time
6	Based on the monitored metrics, the resource elasticity technique dynamically enforces the chunks's resources with PUT /compute_chunk/{compute_chunk_id}/cpus

Figure 25: Overview of resource requests through the Slice Manager API for elastic scaling.

4.3.2.2 Integrating Resource Elasticity with Maestro

Maestro's APIs provide robust capabilities for implementing resource elasticity across various scenarios. For **intra-slice scaling**, the Service Inventory API (TMF 638) can be utilized to dynamically adjust compute, memory, and storage resources for active service instances. This ensures that individual services within a slice can scale vertically to handle fluctuating workloads without disruption. Additionally, for **inter-slice resource management**, Maestro's TMF 641 Service Order Management API enables the orchestrator to allocate or redistribute resources between slices to optimize overall network performance while adhering to SLA requirements. These functionalities allow Maestro to maintain a seamless balance between resource efficiency and service quality, demonstrating its adaptability in both localized and distributed scaling scenarios.

These APIs facilitate communication between the operational support systems (OSS) and business support systems (BSS) within the telecommunications industry. Key features of these APIs are standardization, modularity, ecosystem interoperability, enhanced agility, improved data sharing, etc. The detail of the steps is summarized as follows:

- The first step uses **the Service Ordering API**, which allows for the creation and provisioning of a service with resource specifications for a target infrastructure.
- Upon successful service creation, an organized collection of resources, assets, services and configurations, known as the service inventory, is deployed alongside. This inventory is crucial for managing and tracking the components involved in service delivery.
- For the dynamic adjustment of resources, **the Service Inventory API** can be updated with the PATCH command. This allows changes to be made to the inventory in real-time based on current needs.
- The Management cluster within the Maestro architecture continuously monitors real-time metrics. Based on these metrics, resource allocation for services can be dynamically adjusted to align with network demand, ensuring optimal performance and resource utilization.

Moreover, the decision for resource allocation to each service is determined by MADRL, and it is detailed in the next section.

4.3.3 Computational Resource Elasticity with Multi-Agent Deep Reinforcement Learning

To develop an effective slicing resource management solution, we decided to employ a **MADRL** framework to handle the dynamic allocation of processing power. In this context, the processing power specifically refers to the allocation of CPU resources for different services within the network slice. More specifically, each service within the slice is assigned an independent DRL agent, which is responsible for making real-time decisions about how many CPUs should be allocated to maintain optimal performance. These agents operate independently, yet they jointly make decisions to maximize the performance of the entire system. More specifically, we employ standard MADRL algorithms such as DQN and PPO with a shared reward.

Before analyzing in detail how state and action spaces are characterized, it is appropriate to conduct an analysis on the theoretical foundations of the discipline known as *Markov Games* and one of its major declinations, namely Multi-Agent Reinforcement Learning. After an introduction on these aspects, a state-of-the-art review on MADRL techniques for computational resource management is presented, so to motivate the proposed approach to tackle resource elasticity problems, and particularly the ones in the scope of NANCY.

4.3.3.1 State Space Definition

As seen in the introduction about Markov Games and MARL, the state space is the information that agents use to make the decision, which is based on several key metrics gathered by Prometheus that accurately capture the current of the system and allow for real-time decision-making by the agents. The information included in the state space is summarized in Table 5: Information in the state space.

Table 5: Information in the state space.

Information	Description
Limit	The upper resource boundary allocated to each service.
Usage	The current CPU usage of the service.
Available	The remaining resources available within the slice.
Utilization Percentage	The percentage of allocated resources being utilized by the service.
Other Utilizations	The average resource utilization of other services managed by the multi-agent system within the slice.

This information, also known as **state variables**, provides a comprehensive snapshot of resource allocation and usage, enabling each agent to make informed decisions about how to adjust CPU resources.

4.3.3.2 Shared Reward Function and Utilization Reward

The **shared reward function**[24] is defined as the signal that the agents use to understand how well they perform, it has to be carefully designed to balance efficient resource utilization while maintaining low response times. For example, the shared reward function rewards agents for keeping CPU usage within predefined optimal ranges, and it penalizes both under-utilization and over-utilization. Moreover, the reward function encourages agents to adapt their resource allocations dynamically based on current demand while ensuring the overall performance and responsiveness of the system. Specifically, the

[24] "Reward function," [Online]. Available: https://github.com/sensorlab/agent-edge-autoscaling/blob/51306c1e409ffdbe0d50ffc3bc02cdfaad6c20ec/src/train_mdqn.py#L370

reward combines a **utilization reward**[25], which adjusts based on the proximity of resource usage to the optimal range, and a **shared reward**, which penalizes long response times. This design ensures that agents prioritize both resource efficiency and service quality, driving the system towards optimal performance under varying network conditions. Note that shared reward means that agents are also rewarded for the actions of other agents and their own actions impact on the performance of other agents. Such an approach is well-established in collaborative environments.

4.3.4 Resource Allocation with DQN

With the Deep Q-Network (DQN) algorithm [9], slice resource allocation is performed discretely. In our implementation, the agent controls the service's resource allocation through an **action space** consisting of three possible actions: increase, decrease, or maintain resources. The DQN algorithm uses a deep neural network to approximate Q-values, which represent the expected rewards for each state-action pair. The agent learns an optimal policy through the Q-learning algorithm by updating the Q-values based on observed state-action-reward transitions during interactions with the environment. As a model-free, off-policy algorithm, DQN benefits from experience replay, allowing the agent to learn from past experiences while breaking correlations between consecutive transitions for more efficient training. To further stabilize training, DQN employs a target network that provides fixed Q-value targets, reducing the risk of policy oscillations. This makes DQN suitable for environments where decisions can be mapped to a discrete action space.

4.3.5 Resource Allocation with PPO

In contrast, **Proximal Policy Optimization (PPO)** [10] is designed for **continuous control tasks**, offering finer adjustments compared to discrete approaches like DQN. PPO outputs actions within a continuous range (e.g., [-1, 1]), which can be scaled and applied to adjust resource allocation smoothly and precisely. As a **policy-gradient algorithm**, PPO improves the policy by increasing the likelihood of actions that yield higher rewards. To ensure stability, PPO employs a **clipped surrogate objective**, which limits the size of policy updates, preventing large, destabilizing changes to the policy. This objective closely approximates the true goal of maximizing expected rewards, while the clipping mechanism constrains the amount by which the policy can change in each update, maintaining stable and gradual improvements. Additionally, PPO enhances learning efficiency by reusing the same data multiple times within a single optimization step, extracting more value from each collected experience. This reduces the need for large amounts of data while accelerating the learning process, making it particularly effective in dynamic environments where resources need to be adjusted rapidly. For intra-slice scaling, PPO is advantageous because it allows for more granular adjustments of resources. By selecting a specific range, i.e., the maximum allowable increase or decrease of resources, PPO can allocate resources in a more precise manner compared to the DQN approach, which operates within a discrete action space. This precision is essential for environments that require fine-tuned control of resources to maintain optimal performance.

By utilizing both DQN and PPO in a multi-agent environment, the NANCY resource elasticity module can adapt to a wide range of computing resource management scenarios, such as CPU resource allocation, which will be discussed in the next section.

[25] "Utilization reward," [Online]. Available: https://github.com/sensorlab/agent-edge-autoscaling/blob/51306c1e409ffdbe0d50ffc3bc02cdfaad6c20ec/src/envs.py#L104

4.3.6 Results

4.3.6.1 Simulation Results

In this section, we demonstrate the performance of intra-slice scaling, by deploying a Localization Prediction Service (Localization as a Service, or LaaS), which uses machine learning algorithms to predict device locations based on available data inputs. The service is deployed in a hybrid edge-cloud setup, consisting of a Microk8s cluster running on two Raspberry Pis and a virtual machine. Prometheus Stack is employed to gather and visualize real-time performance metrics. To compare performance, we implement a reactive rule-based scaling approach as a baseline method. More specifically, it predefines thresholds for CPU utilization, which allows us to benchmark the MADRL methods. For instance, when CPU utilization exceeds a specified threshold, the method allocates additional resources. Whereas, when the CPU utilization falls below the threshold, the assigned resources are reduced.

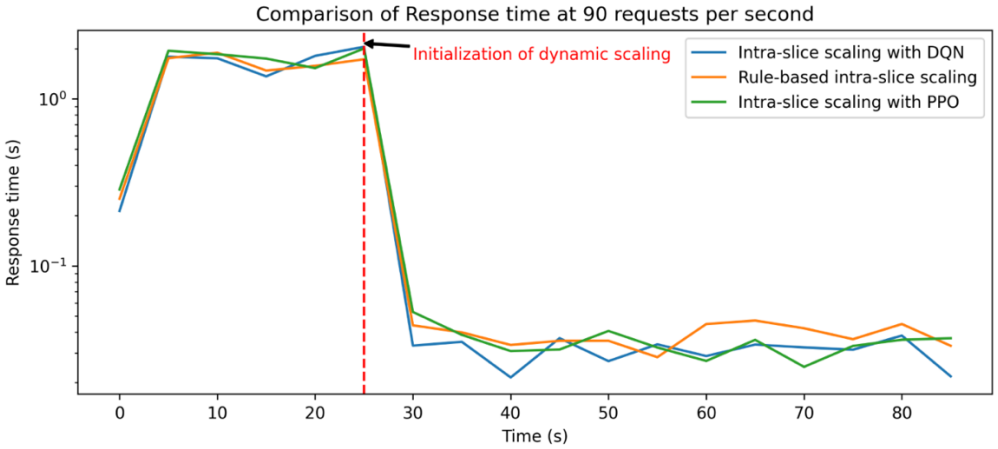


Figure 26: Dynamic Intra-Slice Scaling Methods for Response Time Optimization.

To evaluate the performance of the MADRL-based solutions, response time is considered as one of the main metrics for evaluation, defined as the time interval between requesting a service to make an inference and receiving the results. More specifically, these approaches are compared within a designated time frame and use LaaS for evaluation. In addition, the cluster is subjected to 90 requests per second, and the response time was measured at 5-second intervals. As shown in Figure 26 (on a logarithmic scale), scaling is triggered at the 25-second mark. After fully scaling, the rule-based algorithm has an average of 38.2 ms response time, while PPO had an average of 32.9 ms and DQN had an average of 30.6 ms. Both MADRL methods show slight improvements to the baseline in response times, with DQN providing a 20% improvement and PPO delivering a 14% improvement.

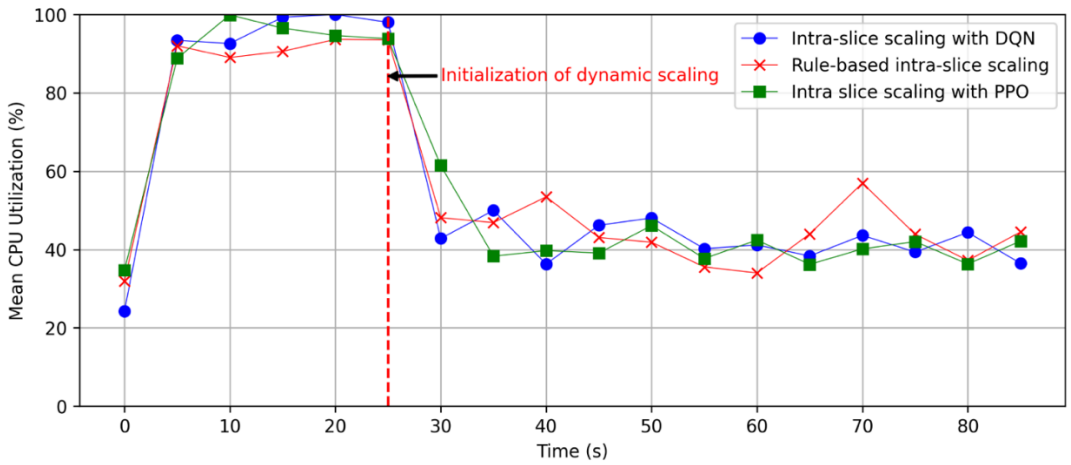


Figure 27: Efficiency of Dynamic Intra-Slice Scaling Methods.

CPU Utilization is another performance metric for evaluating the effectiveness of the intra-slice scaling algorithms. By using the same setup, Figure 27 illustrates the average CPU utilization of all services in the cluster over time, demonstrating the effectiveness of the MADRL-based methods. As we can see from the figure, PPO delivers a smoother and more consistent utilization pattern, while the other methods exhibit more abrupt fluctuations in resource allocation. This is due to PPO having more fine-grained resource adjustment than DQN as well as rule-based methods. This is further supported by Figure 28, Figure 29 and Figure 30, where the CPU allocations and utilization are displayed as service scale over the designated time frame. As depicted in Figure 30, PPO shows lees fluctuation of the CPU allocations than the other two methods, which implies granular adjustments made by PPO result in a more efficient scaling process, as evidenced by the steadier resource usage compared to the abrupt changes seen in the discrete methods.

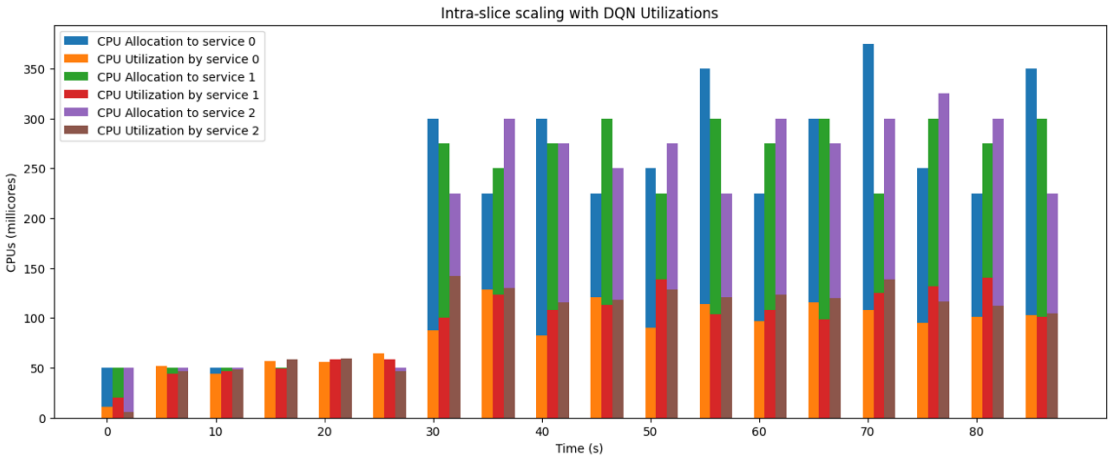


Figure 28: Resource allocation and utilization of Dynamic Intra-Slice Scaling with DQN.

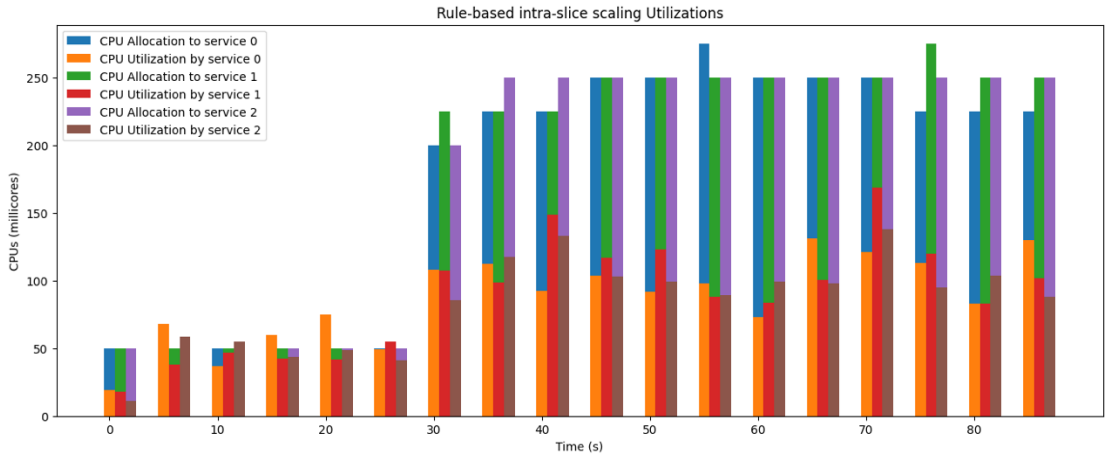


Figure 29: Resource allocation and utilization of Dynamic Intra-Slice Scaling with Rule-based.

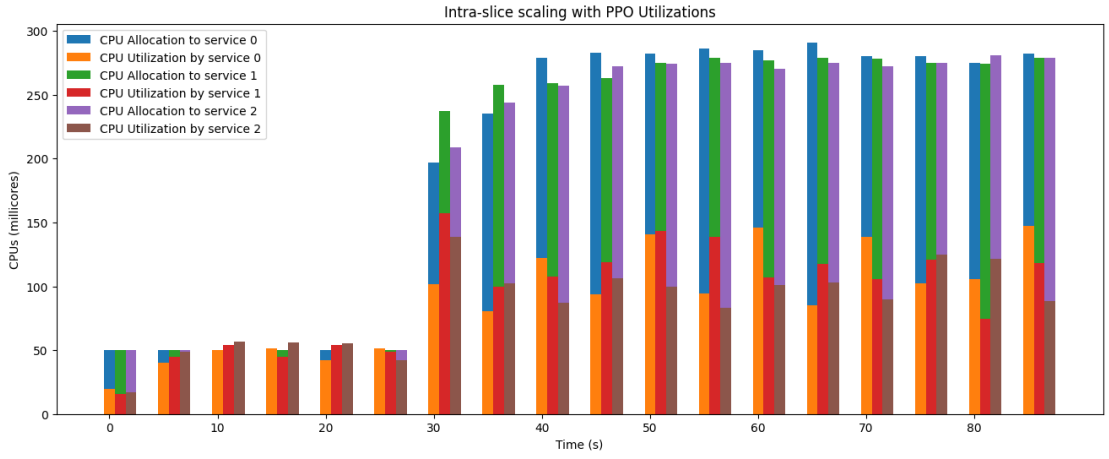


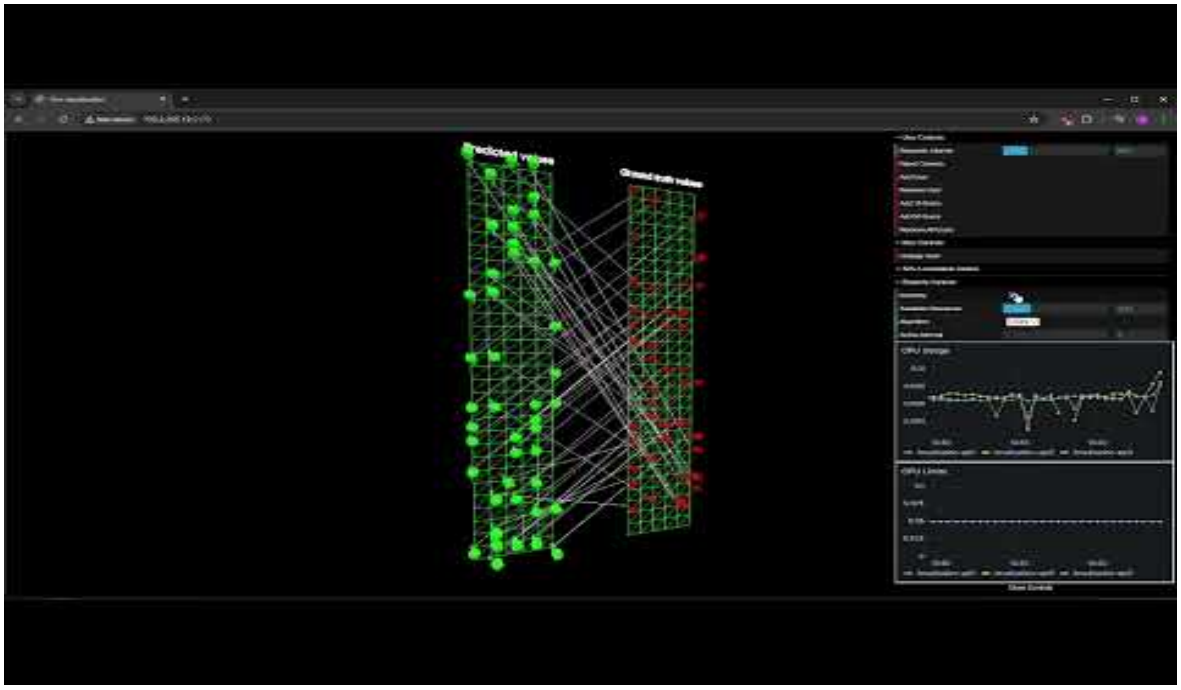
Figure 30: Resource allocation and utilization of Dynamic Intra-Slice Scaling with PPO.

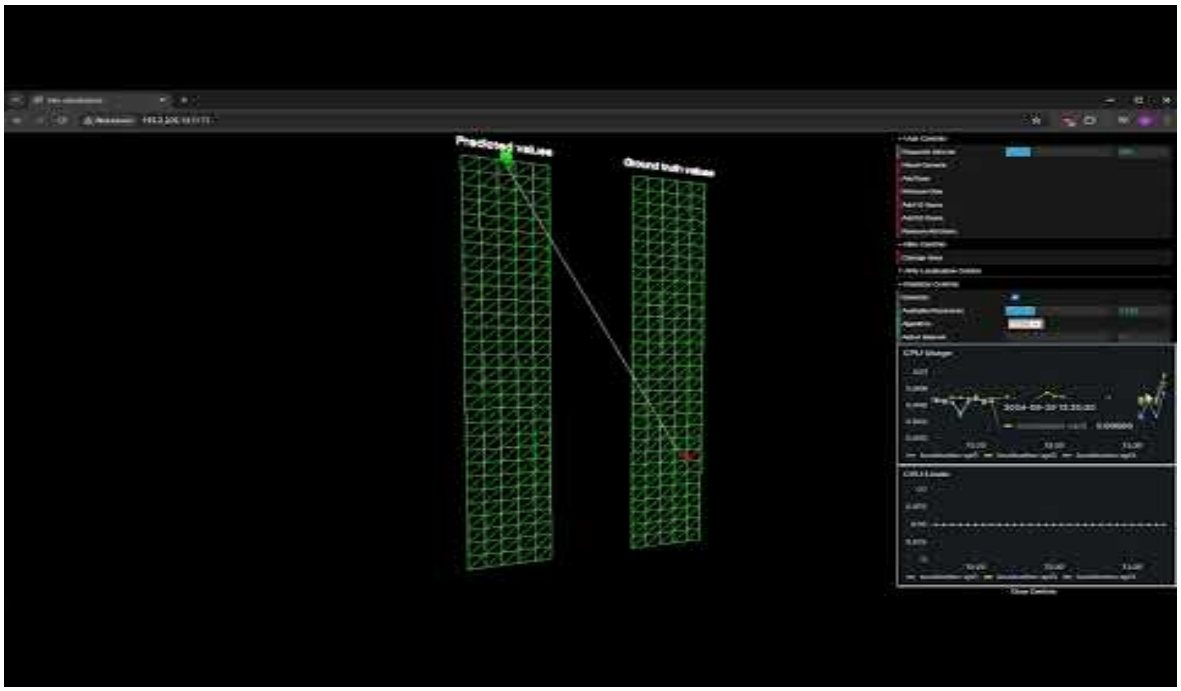
4.3.6.2 Demonstration

The localization service from D3.3 and the DRL-based scaling implementations were integrated into a live demonstration to showcase real-time resource elasticity managed by DRL agents. During the demonstration, system performance was analyzed both before and after resource elasticity was applied. Without elasticity, inadequate scaling led to longer response times and a clear degradation in performance. Once resource elasticity was introduced, the DRL agents dynamically adjusted resources

according to network demands, significantly improving response times and overall system performance. This dynamic scaling capability highlights the advantages of DRL in managing complex, real-time resource allocation scenarios effectively.

A video demonstration of the real-time resource elasticity managed by DRL agents has been uploaded at the following link: https://youtu.be/mIydIkWhcoI





The source code will be available at the following repository: https://github.com/sensorlab/agent-edge-autoscaling

5 Conclusion

The advancements presented in this deliverable highlight NANCY's commitment to achieving flexible resource management and cost efficiency to satisfy the QoS/QoE for the applications and the users in the dynamic network demands. By addressing the complexity of distributed resource scaling. This deliverable underscores the role of elasticity in optimizing computational and network resources to ensure high performance across diverse applications and scenarios.

Two orchestrators, Slice Manager and Maestro, enable these resource elasticity techniques through advanced virtualization technologies that dynamically adjust resources based on demands through APIs.

The development of resource elasticity techniques within the framework of NANCY has been presented. Namely, SCHED_DEADLINE, PHaul, and MADRL-based computational scaling. Each technique has its own purpose and function. For example, SCHED_DEADLINE ensures predictable CPU bandwidth allocation for time-sensitive applications, maintaining low latency and stable performance across high-demand periods. PHaul, a DRL-based path allocation mechanism, dynamically allocates network resources across paths to meet latency, throughput, and bandwidth requirements, effectively managing network load and enhancing resilience. Lastly, the MADRL-based computational resource elasticity enables real-time scaling of CPU and memory within slices, allowing seamless adaptation to fluctuating user demands while optimizing resource utilization. Moreover, these techniques have been validated, exhibiting their feasibility in real-world scenarios. Implementation details, including code snippets and open source repositories and a demo together with experimental results, are provided to facilitate further development, integration, and adoption.

Bibliography

[1] A. Kwan, J. Wong, H.-A. Jacobsen and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," IEEE 39th International Conference on Distributed Computing Systems, 2019, pp. 80-90.

[2] L. E. Zachrisson, "Markov games," Advances in game theory, 52, 1964, pp. 211-253.

[3] G. Chalkiadakis, E. Elkind and M. Wooldridge, “Computational aspects of cooperative game theory”, Springer Nature, 2022.

[4] R. Song, Q. Wei, H. Zhang and F. L. Lewis, "Discrete-time non-zero-sum games with completely unknown dynamics," IEEE Transactions on Cybernetics, vol. 51, no. 6, 2019, pp. 2929-2943.

[5] M. A. Amaral, L. Wardil, M. Perc and J. K. da Silva, "Evolutionary mixed games in structured populations: Cooperation and the benefits of heterogeneity," Physical Review, vol. 93, no. 4, pp. 042304, 2016.

[6] C. Daskalakis, P. W. Goldberg and C. H. Papadimitriou, "The Complexity of Computing a Nash Equilibrium," Communications of the ACM, vol. 52, no. 2, pp. 89-97, 2009.

[7] W. W. Cohen and H. Hirsh, "Markov games as a framework for multi-agent reinforcement learning," Machine Learning Proceedings, 1994, pp. 157-163.

[8] D. Zeng, L. Gu, S. Pan, J. Cai and S. Guo, "Resource management at the network edge: A deep reinforcement learning approach," IEEE Network, vol. 33, no. 3, pp. 26-33, 2019.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," 2013.

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal policy optimization algorithms," 2017.

[11] A. Alwarafy, M. Abdallah, B. S. Çiftler, A. Al-Fuqaha and M. Hamdi, "The frontiers of deep reinforcement learning for resource management in future wireless HetNets: Techniques, challenges, and research directions," IEEE Open Journal of the Communications Society, vol. 3, pp. 322-365, 2022.

[12] W. Miao, Z. Zeng, M. Zhang, S. Quan, Z. Zhang, S. Li and Q. Sun, "Multi-agent reinforcement learning for edge resource management with reconstructed environment," IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, 2021, pp. 1729-1736.

[13] N. Naderializadeh, J. Sydir, J. M. Simsek and H. Nikopour, "Resource management in wireless networks via multi-agent deep reinforcement learning," IEEE Transactions on Wireless Communications, vol. 20, no. 6, pp. 3507-3523, 2021.

[14] Y. Nie, J. Zhao, F. Gao and F. R. Yu, "Semi-distributed resource management in UAV-aided MEC systems: A multi-agent federated reinforcement learning approach," IEEE Transactions on Vehicular Technology, vol. 70, no. 12, pp. 13162-13173, 2021.

[15] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z. T. Kalbarczyk and R. K. Iyer, "Reinforcement learning for resource management in multi-tenant serverless platforms," 2nd European Workshop on Machine Learning and Systems, 2022, pp. 20-28.

[16] D. Wang, B. Li, B. Song, Y. Liu, K. Muhammad and X. Zhou, "Dual-driven resource management for sustainable computing in the blockchain-supported digital twin IoT," IEEE Internet of Things Journal, vol. 10, no. 8, pp. 6549-6560, 2022.

[17] X. Du, T. Wang, Q. Feng, C. Ye, T. Tao, L. Wang and M. Chen, "Multi-agent reinforcement learning for dynamic resource management in 6G in-X subnetworks," IEEE Transactions on Wireless Communications, vol. 22, no. 3, pp. 1900-1914, 2022.

[18] H. Wu, D. Qiu, L. Zhang and M. Sun, "Adaptive multi-agent reinforcement learning for flexible resource management in a virtual power plant with dynamic participating multi-energy buildings," Applied Energy, vol. 374, p. 123998, 2024.

[19] M. Ahmed, J. Liu, M. A. Mirza, W. U. Khan and F. N. Al-Wesabi, "MARL based resource allocation scheme leveraging vehicular cloudlet in automotive-industry 5.0," Journal of King Saud University-Computer and Information Sciences, vol. 35, no. 6, p. 101420, 2023.

[20] J. Rosenberger, M. Urlaub, F. Rauterberg, T. Lutz, A. Selig, M. Bühren and D. Schramm, "Deep reinforcement learning multi-agent system for resource allocation in industrial internet of things," Sensors, vol. 22, no. 11, p. 4099, 2022.

[21] H. Zhang, C. Lu, H. Tang, X. Wei, L. Liang, L. Cheng and Z. Han, "Mean-field-aided multiagent reinforcement learning for resource allocation in vehicular networks," IEEE Internet of Things Journal, vol. 10, no. 3, pp. 2667-2679, 2022.

[22] TMForum, “TMF633 Service Catalog Management API v4.0.0,” 2021, [Online]. Available: https://www.tmforum.org/resources/standard/tmf633-service-catalog-api-user-guide-v4-0-0/

[23] TMForum, “TMF641 Service Ordering Management API v4.1.1,” 2021, [Online]. Available: https://www.tmforum.org/resources/specifications/tmf641-service-ordering-management-api-user-guide-v4-1-1/

[24] TMForum, “TMF638 Service Inventory Management API v4.0.1,” 2020, [Online]. Available: https://www.tmforum.org/resources/specification/tmf638-service-inventory-api-user-guide-v4-0-0/

[25] TMForum, “TMF634 Resource Catalog Management API v4.1.0,” 2021, [Online]. Available: https://www.tmforum.org/resources/specification/tmf634-resource-catalog-management-api-user-guide-v4-1-0/

[26] TMForum, “TMF652 Resource Ordering Management API v4.0.0,” 2020, [Online]. Available: https://www.tmforum.org/resources/specification/tmf652-resource-order-management-api-user-guide-v4-0-0/

[27] TMForum, “TMF639 Resource Inventory Management API v4.0.0,” 2020, [Online]. Available: https://www.tmforum.org/resources/specification/tmf639-resource-inventory-api-user-guide-v4-0/

[28] TMForum, “TMF632 Party Management API v5.0.0,” 2023, [Online]. Available: https://www.tmforum.org/resources/specifications/tmf632-party-management-api-rest-specification-v5-0-0/

[29] TMForum, “TMF669 Party Role Management API v5.0.0,” 2023, [Online]. Available: https://www.tmforum.org/resources/specifications/tmf669-party-role-management-api-user-guide-v5-0-0/

[30] TMForum, “TMF674 Geographic Site Management API v4.0.1,” 2020, [Online]. Available: https://www.tmforum.org/resources/specification/tmf674-geographic-site-management-api-user-guide-v4-0/

[31] TMForum, “TMF673 Geographic Address Management API v4.0.0,” 2020, [Online]. Available: https://www.tmforum.org/resources/specification/tmf673-geographic-address-management-api-user-guide-v4-0-0

[32] C. Tranoris, "OpenSlice: An opensource OSS for delivering network slice as a service," arXiv:2102.03290, Feb. 2021.

[33] J. Lelli, C. Scordino, L. Abeni and D. Faggioli, "Deadline scheduling in the Linux kernel," Software: Practice and Experience, vol. 46, no. 6, p. 821–839, 2016.

[34] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," 19th IEEE Real-Time Systems Symposium, 1998, pp. 4-13.

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

[36] T. Cucinotta, F. Checconi, L. Abeni and L. Palopoli, "Self-tuning schedulers for legacy real-time applications," 5th European Conference on Computer Systems, 2010, p. 55–68.

[37] T. Cucinotta, F. Checconi, L. Abeni and L. Palopoli, "Adaptive real-time scheduling for legacy multimedia applications," ACM Transactions on Embedded Computing Systems, vol. 11, no. 4, 2013.

[38] L. Abeni, T. Cucinotta and D. Casini, "Period Estimation for Linux-based Edge Computing Virtualization with Strong Temporal Isolation," 3rd Real-time And intelliGent Edge computing workshop, Hong Kong, 2024.

[39] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," Journal of the ACM (JACM), vol. 20, no. 1, pp. 46-61, 1973.

[40] L. Abeni, A. Biondi and E. Bini, "Partitioning real-time workloads on multi-core virtual machines," Journal of Systems Architecture, vol. 131, p. 102733, 2022.

[41] L. Abeni, A. Biondi and E. Bini, "Hierarchical Scheduling of Real-Time Tasks over Linux-based Virtual Machines," Journal of Systems and Software, vol. 149, 2019.

[42] 3GPP, "Energy Efficiency; Solutions for Network Energy Saving," 2010.

[43] J. Pueyo, D. Camps-Mur and M. Catalan-Cid, "PHaul: A PPO-Based Forwarding Agent for Sub6 Enhanced Integrated Acess and Backhaul Networks," in Proc. IEEE Transactions on Network and Service Management.

[44] W. Yuhui, H. Hao, W. Chao and T. Xiaoyang, "Truly Proximal Policy Optimization," 2019.

[45] M. Cudak, A. Ghosh and J. Andrews, "Integrated Accessand Backhaul: A Key Enabler for 5G Millimeter-Wave Deployments," in IEEE Communications Magazine, vol. 59, no. 4, 2021, pp. 88-94.

[46] Fundacio-i2CAT, "PHAUL, a Deep Reinforcement Learning Agent that produces the optimal flow allocations for Integrated Access Backhaul networks," [Online]. Available: https://github.com/Fundacio-i2CAT/PHaul