Green Technologies for 5/6G Service-Based Architecture

Grant Agreement No	101096925
Full Title	Green Technologies for 5/6G Service-Based Architectures
Start date	01 Jan 2023
End date	30 Apr 2026
Duration	40
Project URL	https://www.6green.eu
Coordinator	CONSORZIO NAZIONALE INTERUNIVERSITARIO PER LE TELECOMUNICAZIONI (CNIT)

Deliverable D2.3

The 6Green Enabling Technologies

Contractual due date	31/12/2025 (M36)	Actual submission date	16/01/2026
Nature	R - Report	Dissemination Level	PU - Public
Lead Beneficiary	ICCS		
Responsible Author	Anastasios Zafeiropoulos, Eleni Stai, Nikos Fryganiotis, Petros Maratos (ICCS)		
Contributions from	Marius Iordache, Catalin Brezeanu, Daniel Tichiu (ORO); Daniele Ronzani (HPE); Chiara Lombardo, Paolo Bono, Davide Freggiaro, Alderico Gallo, Nicole S. Martinelli, Ramin Rabbani, Beatrice Siccardi (CNIT); Jonathan Rivalan, Hai Long Ngo (SMILE); Jane Frances Pajo (TNOR); Janez Sterle, Luka Koršič, Rudolf Sušnik (ININ); Javier Velázquez Martínez, Luis Miguel Contreras Murillo, Guillermo Sánchez Illán (TID); Claudio Cicconetti, Raffaele Bruno (CNR); Orazio Toscano (TEI).		

Revision history

Version	Issue Date	Changes	Contributor(s)
v0.1	01/07/2025	Initial version	Anastasios Zafeiropoulos, Eleni Stai, Nikos Fryganiotis (ICCS)
v0.2	15/09/2025	Revision of contributions in Sections 2-5	Anastasios Zafeiropoulos, Eleni Stai, Nikos Fryganiotis, Petros Maratos (ICCS); Marius Iordache, Catalin Brezeanu, Daniel Tichiu (ORO); Daniele Ronzani (HPE); Chiara Lombardo, Paolo Bono, Davide Freggiaro, Alderico Gallo, Nicole S. Martinelli, Ramin Rabbani (CNIT); Jonathan Rivalan, Hai Long Ngo (SMILE); Jane Frances Pajo (TNOR); Janez Sterle, Luka Koršič, Rudolf Sušnik (ININ); Javier Velázquez Martínez, Luis Miguel Contreras Murillo, Guillermo Sánchez Illán (TID); Claudio Cicconetti, Raffaele Bruno (CNR); Orazio Toscano (TEI).
v0.3	20/10/2025	Finalization of contributions in Sections 2-5	Anastasios Zafeiropoulos, Eleni Stai, Nikos Fryganiotis, Petros Maratos (ICCS); Marius Iordache, Catalin Brezeanu, Daniel Tichiu (ORO); Daniele Ronzani (HPE); Chiara Lombardo, Paolo Bono, Davide Freggiaro, Alderico Gallo, Nicole S. Martinelli, Ramin Rabbani (CNIT); Jonathan Rivalan, Hai Long Ngo (SMILE); Jane Frances Pajo (TNOR); Janez Sterle, Luka Koršič, Rudolf Sušnik (ININ); Javier Velázquez Martínez, Luis Miguel Contreras Murillo, Guillermo Sánchez Illán (TID); Claudio Cicconetti, Raffaele Bruno (CNR); Orazio Toscano (TEI).
v0.4	15/11/2025	Final contributions in all the sections	Anastasios Zafeiropoulos, Eleni Stai, Nikos Fryganiotis, Petros Maratos (ICCS); Marius Iordache, Catalin Brezeanu, Daniel Tichiu (ORO); Daniele Ronzani (HPE); Chiara Lombardo, Paolo Bono, Davide Freggiaro, Alderico Gallo, Nicole S. Martinelli, Ramin Rabbani, Beatrice Siccardi (CNIT); Jonathan Rivalan, Hai Long Ngo (SMILE); Jane Frances Pajo (TNOR); Janez Sterle, Luka Koršič, Rudolf Sušnik (ININ); Javier Velázquez Martínez, Luis Miguel Contreras Murillo, Guillermo Sánchez Illán (TID); Claudio Cicconetti, Raffaele Bruno (CNR); Orazio Toscano (TEI).
v0.5	07/12/2025	Complete draft for internal review	Anastasios Zafeiropoulos, Eleni Stai, Nikos Fryganiotis, Petros Maratos (ICCS);
v0.6	12/12/2025	Internal review	Chiara Lombardo (CNIT); Luis Miguel Contreras Murillo (TID)
v0.7	19/12/2025	Revisions upon internal review	All contributors
v1.0	16/01/2026	Version ready for submission	Chiara Lombardo, Riccardo Rapuzzi (CNIT)

Table of Contents

List of Figures

List of Tables

Glossary of terms and abbreviations used

Abbreviation / Term	Description
AF	Application Function
AMF	Access and Mobility Management Function
CCMF	Cloud-Continuum Management Function
CRD	Custom Resource Definition
DAG	Directed Acyclic Graph
DNN	Data Network Name
eBPF	extended Berkeley Packet Filter
EdgeMF	Edge-cloud Management Function
FaaS	Function-as-a-Service
GRU	Gated Recurrent Unit
HPA	Horizontal Pod Autoscaler
MaaS	Metal-as-a-Service
NEF	Network Exposure Function
NRP	Network Resource Partition
NSC	Network Slice Controller
PCF	Policy Control Function
SBA	Service Based Architecture
SBI	South-Bound Interface
SLO	Service Level Objective
SMF	Session Management Function
S-NSSAI	Single – Network Slice Selection Assistance Information
TSD	Time Series Database
UDM	Unified Data Management
UPF	User Plane Function
URSP	UE Route Selection Policy
VIM	Virtual Infrastructure Manager
VRF	Virtual Routing and Forwarding
WIM	Wide-Area Infrastructure Manager
XDP	eXpress Data Path
NSI	Network Slice instance
EGMF	Exposure Governance Management Function

Abbreviation / Term	Description
NSMF	Network Slice Management Function
NFMF	Network Function Management Function
NFVO	Network Functions Virtualization Orchestrator
VNF	Virtualized Network Function
CNF	Containerized Network Function

Executive Summary

This deliverable details the final version of the work realized within 6Green towards the specification of a set of enabling technologies that are adopted towards the development of the 6Green Service Based Architecture (SBA) in WP3, as well as towards the development of vertical application orchestration mechanisms in WP4. It builds upon the results provided in the D3.2 that detailed the work in progress in the implementation of these technologies in M18 of the project.

The set of enabling technologies include:

- network connectivity management and traffic offloading mechanisms,
- cloud-native orchestration mechanisms considering approaches that take advantage of service-mesh techniques, as well as automation mechanisms based on infrastructure as a code, ZeroOps and continuous automation principles,
- power management mechanisms for the core, transport and access part of the continuum by considering serverless workloads,
- network slice lifecycle management and optimization techniques, including energy-aware network slice management in O-RAN and multi-provider settings, and
- green observability and profiling mechanisms.

For all cases, the final status of the development of the enablers is provided. A mapping of the enabling technologies with their adoption toward the development of the various functions of the 6Green SBA is also detailed, while information for the developed software prototypes is made available in D2.4.

1 Introduction

This document details the activities carried out in the Work Package 2 (WP2), “Green Enabling Technologies for Cloud-Native Services” of the 6Green Project. The goal of the 6Green Project is to create an innovative, service-based, and comprehensive ecosystem that expands the communication infrastructure into a sustainable, interconnected, and greener end-to-end inter-computing system. It aims to promote energy efficiency across the entire 5/6G value chain and reduce the carbon footprint of 5/6G networks and vertical applications.

The 6Green Project is structured around three main research axes, which correspond to three administrative domains/layers within the architecture: the **5/6G Edge-Cloud Infrastructure**, the **Network Platform** and the **Vertical Application** domains. The corresponding research areas are referred to as “Enabling Technologies for Cloud-Native Service Meshes,” “the 6Green Service-based Architecture,” and “Vertical Application Orchestration within the 5/6G Green Economy.” These axes must closely collaborate to implement the holistic vision of 6Green.

WP2 primarily focused on the activities related to the first research axis of the project: “Green Enabling Technologies for Cloud-Native Services.” Within WP2, the initial activities involved refining the architectural definition of the 6Green ecosystem and validating use cases for the technologies and solutions developed by the project. This includes identifying the roles of different stakeholders, determining system and use cases requirements, and establishing key performance indicators for the identified use cases.

Following, effort was allocated into the development of a set of enabling technologies to boost green elasticity (automatically, and rapidly provision, adapt, and de-provision network and (edge) computing resources/artefacts and hardware offload engines to improve energy efficiency) and edge agility (transparently move applications/services (or part of them) at run-time in different geographical areas of the edge-cloud continuum) in the deployment of services and applications over a 6G infrastructure. These enabling technologies include network connectivity management and traffic offloading mechanisms; cloud-native orchestration mechanisms considering approaches that take advantage of service-mesh techniques, as well as automation mechanisms based on infrastructure as a code, ZeroOps and continuous automation principles; power management mechanisms for the core, transport and access part of the continuum by considering serverless workloads; network slice lifecycle management and optimization techniques, including energy-aware network slice management in O-RAN and multi-provider settings, and green observability and profiling mechanisms.

In the current document, the overall work towards the development of such enabling technologies is detailed. The document builds upon the results provided in the D3.2 that detailed the work in progress in the implementation of these technologies in M18 of the project. In all cases, the final status of the development of the enablers is provided. A mapping of the enabling technologies with their adoption toward the development of the various functions of the 6Green SBA is also detailed, while information for the developed software prototypes is made available in D2.4.

Per enabling technology, the design of the mechanisms, the implementation status and evaluation results are provided. With this objective in mind, this document is organized as follows. Section 2 details the connectivity and traffic offloading enablers for managing data traffic to optimize performance, reduce congestion, and enhance the user experience across resources in the computing continuum. Section 3 details enablers that support orchestration actions for the management of cloud-native software, including enablers that support automation in the management of compute and network infrastructure in 5G/6G environments. Section 4 details power management and network slice lifecycle management techniques by considering serverless workloads. Section 5 details data fusion, profiling and observability mechanisms. Section 6 provides a short mapping of the detailed enabling technologies with the 6Green Service-based Architecture (SBA), while Section 7 concludes the document.

2 Connectivity and Traffic Offloading

2.1 Traffic Offloading to Hardware Acceleration-Based UPF by Control Plane Service Exposure

Traffic offloading in 5/6G networks refers to the process of redirecting data traffic to alternative pathways to optimize performance, reduce congestion, enhance the user experience, or change user plane technology. Traffic offloading is a crucial feature for allowing Green Elasticity, as the energy-aware usage of hardware accelerators cannot be applied unless traffic is redirected accordingly.

During the 6Green activities, traffic offloading has been designed and implemented through the development of an evolved Network Exposure Function (NEF) prototype. Initially introduced in D2.2 and further detailed in this section, the prototype is tailored specifically for the 6Green SBA. It implements the traffic offloading mechanism in alignment with the 3GPP standard, particularly referencing TS 29.502 and TS 29.522 for the API specifications. The traffic offloading procedure leverages the URSP rule modification applied within the Service parameter provisioning service, as defined in D2.2. However, to better support the needs of 6Green, this service has been extended and slightly adapted to offer greater flexibility for slice offloading, by working on **slice change** for a set of UEs.

Unlike the approach described in Section 2.1.1 of D2.2, the service in 6Green is no longer referred to as *Nnef_ServiceParameter*. Instead, it has been evolved and renamed as ***Nnef_SliceOffloading***, to more accurately reflect its role of changing slice for specific UEs, within the 6Ggreen framework. In the next section, further detail about the implementation is provided.

2.1.1 Slice Offloading Mechanism within the 5GC

The legacy 3GPP standardized Service Parameter provisioning mechanism in 5GS enables external entities to supply service-specific parameters for single UE, facilitating its traffic steering. This is particularly effective when combined with (AF)-guided URSP rules, which allow for tailored routing of third-party application traffic. As described in the previous section, in the context of 6Green Project, this mechanism has been enhanced to introduce dynamic traffic steering capabilities.

The innovation lies in the ability to incorporate new input parameters such as the originating network slice or DNN, allowing multiple UEs belonging to such domains (S-NSSAI or DNN) to be moved from a slice to another one. This is possible exploiting the availability of management APIs exposed by 5G Core Networks. In this activity we utilized the HPE Aruba Networking Private 5GC, which provides dedicated provisioning APIs allowing configuration changes (e.g., slice assignment for UE profiles).

The developed Nnef_SliceOffloading service leverages some information coming from the CN and the actual request from the service consumer.

Figure 2-1 illustrates the complete flowchart of the functional slice offloading procedure. The process requires two main inputs: information from the client request (i.e., destSliceList and target) and the provisioning data from the core network (i.e., profileList and subscriberList). The first step involves extracting the actual destination slice (destSlice), and related provisioning profile (destProfile) among the candidate ones included in destSliceList. The second step focuses on identifying the target UE or set of UEs. Based on the target type (GroupID, Slice, supi, gpsi, ipv4) included in the request, the algorithm selects the appropriate procedure to obtain the list of UEs (or a single UE) whose profiles need to be updated with the one including the destination slice. If a UE already has a profile that includes the destination slice, it is simply excluded from further processing.

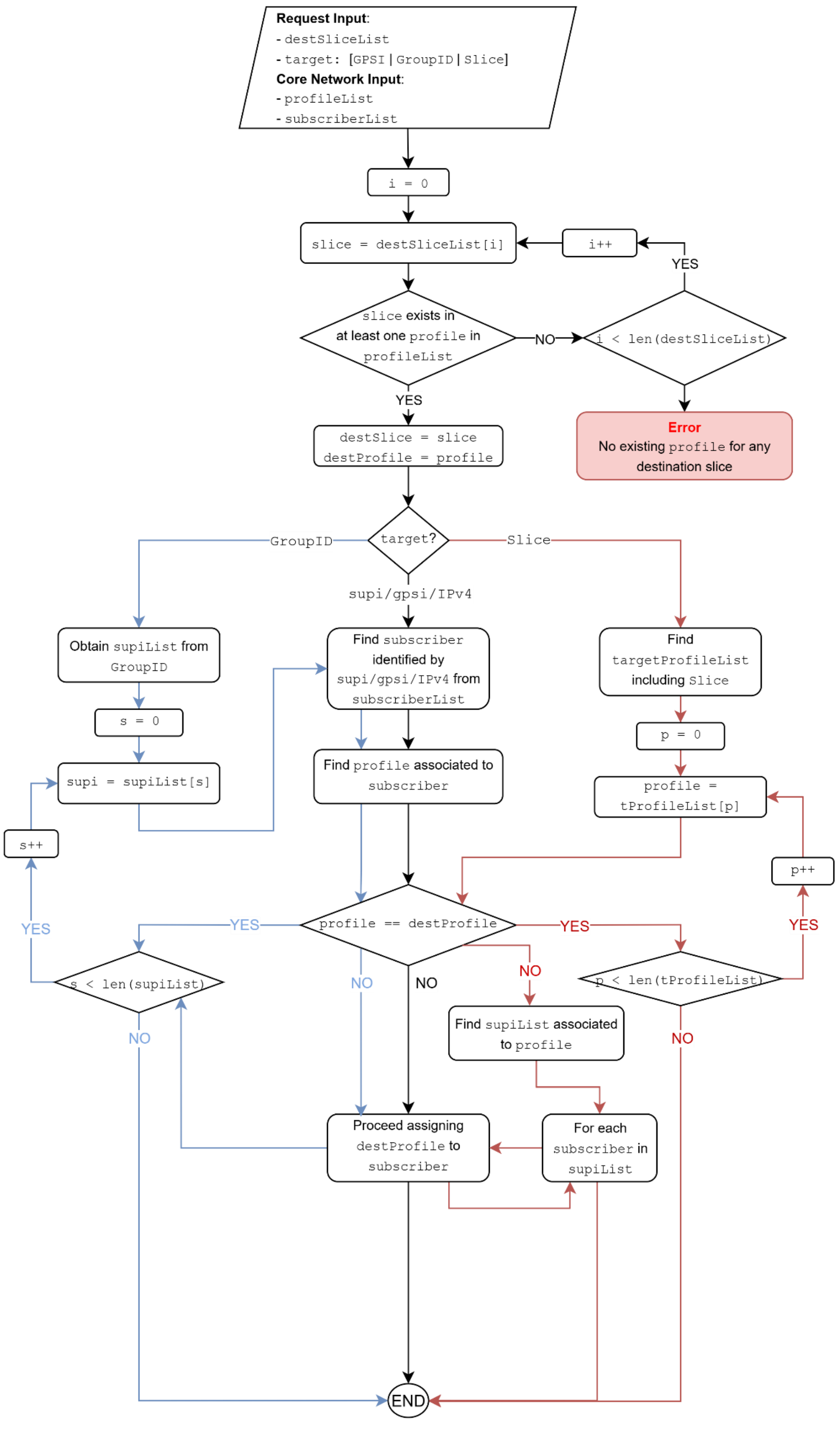

Request Input:
- destSliceList
- target: [GPSI | GroupID | Slice]
Core Network Input:
- profileList
- subscriberList
i = 0
slice = destSliceList[i]
i++
YES
slice exists in at least one profile in profileList
NO
i < len(destSliceList)
YES
destSlice = slice
destProfile = profile
Error
No existing profile for any destination slice
target?
GroupID
Slice
supi/gpsi/IPv4
Obtain supiList from GroupID
s = 0
supi = supiList[s]
s++
Find subscriber identified by supi/gpsi/IPv4 from subscriberList
Find profile associated to subscriber
Find targetProfileList including Slice
p = 0
profile = tProfileList[p]
p++
profile == destProfile
YES
NO
s < len(supiList)
p < len(tProfileList)
Find supiList associated to profile
Proceed assigning destProfile to subscriber
For each subscriber in supiList
END

Figure 2-1: Flowchart representation of the Slice offloading procedure.

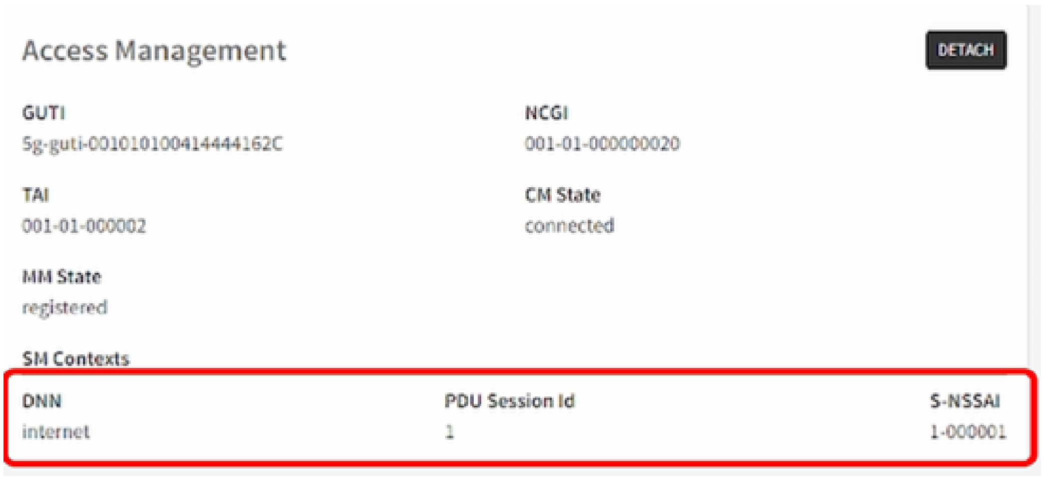
2.1.2 Validation

The slice offloading procedure was evaluated using a single User UE. The tested Core Network was based on the HPE Aruba Networking Private 5G Core (HPE 5GC).

The UE's data was provisioned in the 5GC and initially assigned to Profile 1, associated with Slice 1-000001. A second profile (Profile 2) was configured and linked to Slice 1-000002. Each slice was mapped to a dedicated UPF, enabling traffic redirection upon slice switching. UERANSIM was used to emulate both the gNB and the UE, with the gNB configured to support both slices.

Phase 1: UE registration and PDU session establishment to Slice 1

As shown in Figure 2-2, the UE has been successfully connected to the initial Slice 1. A ping test was performed to check the connectivity and traffic.



Access Management

DETACH

GUTI
5g-guti-001010100414444162C

NCGI
001-01-000000020

TAI
001-01-000002

CM State
connected

MM State
registered

SM Contexts

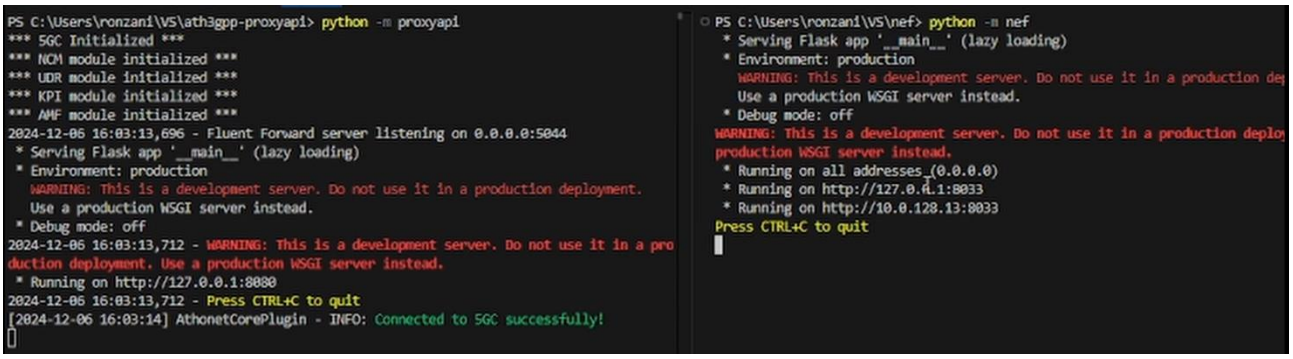
DNN	PDU Session Id	S-NSSAI
internet	1	1-000001

Figure 2-2: AMF information regarding UE connectivity status.

Phase 2: Execution of NEF and tools for interfacing with the 5GC

Figure 2-3 illustrates the execution of the software prototypes. On the right side, the NEF Python application, including the SliceOffloading service, is show. On the left, the ProxyAPI component is depicted, acting as an interface layer to communicate with the 5GC APIs.

The NEF application is configured to interact with the ProxyAPI when API access is required. The ProxyAPI is already bound to the 5G Core, as indicated by the green status confirmation.



```
PS C:\Users\ronzani\VS\ath3gpp-proxyapi> python -m proxyapi
*** 5GC Initialized ***
*** NCM module initialized ***
*** UDR module initialized ***
*** KPI module initialized ***
*** AMF module initialized ***
2024-12-06 16:03:13,696 - Fluent Forward server listening on 0.0.0.0:5044
 * Serving Flask app '__main__' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
2024-12-06 16:03:13,712 - WARNING: This is a development server. Do not use it in a pro
duction deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:8080
2024-12-06 16:03:13,712 - Press CTRL+C to quit
[2024-12-06 16:03:14] AthonetCorePlugin - INFO: Connected to 5GC successfully!
```

```
PS C:\Users\ronzani\VS\nef> python -m nef
 * Serving Flask app '__main__' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production de
   Use a production WSGI server instead.
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deplo
production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8033
 * Running on http://10.0.128.13:8033
Press CTRL+C to quit
```

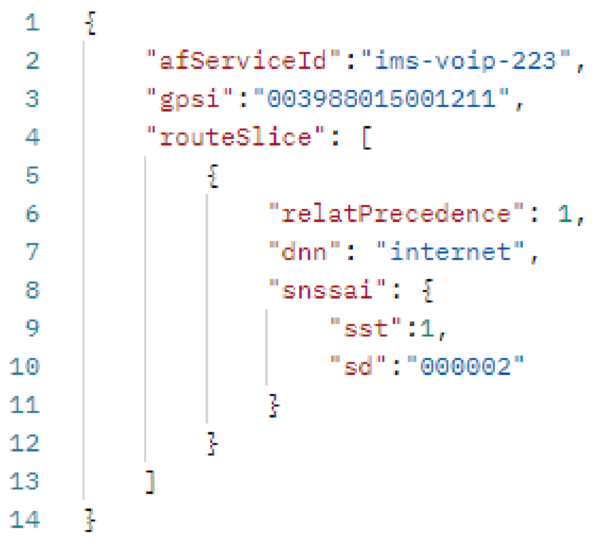
Figure 2-3: Terminal output with execution logs of ProxyAPI (left) and NEF prototype (right).

Phase 3: Slice offloading request from client

The functional test proceeded with the execution of the slice switch request using Postman. The request body (see Figure 2-4) includes the essential parameters required to trigger the slice transition:

- **GPSI**, used to uniquely identify the target UE
- **Route slice information**, specifying the combination of DNN and S-NSSAI values, which will be mapped to an available destination profile to be assigned to the UE.

These parameters were processed by the NEF application to initiate the slice offload procedure for the selected UE.



```
{
    "afServiceId":"ims-voip-223",
    "gpsi":"003988015001211",
    "routeSlice": [
        {
            "relatPrecedence": 1,
            "dnn": "internet",
            "snssai": {
                "sst":1,
                "sd":"000002"
            }
        }
    ]
}
```

Figure 2-4: SliceOffload request body.

Phase 4: SliceOffloading procedure and results

Once the request was triggered, the SliceOffloading service executed the slice switching logic by interacting with the 5G Core through the ProxyAPI component. Upon successful assignment of Profile 2 to the UE, subsequent user traffic was redirected through the second UPF, as defined by the new slice configuration.

Figure 2-5 shows the traffic handled by the two UPFs, visualized through Grafana plots. The transition of traffic from Slice 1 to Slice 2 is clearly observable, confirming the effectiveness of the offloading procedure.

In order to activate the new slice configuration, the UE was detached and subsequently re-attached. This procedure is necessary to ensure that Profile 2 was correctly applied, enabling user traffic to be routed through the second UPF associated with Slice 2.

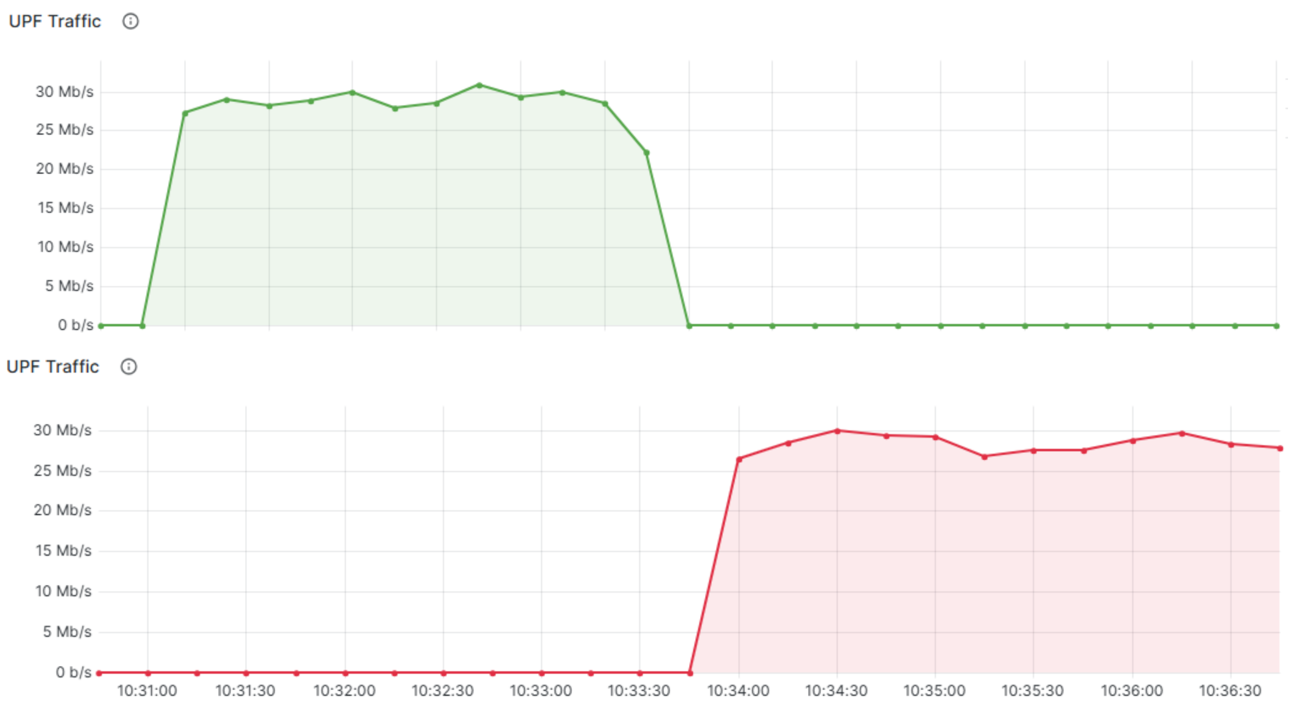


Figure 2-5: UPF 1 (above) and UPF 2 (below) user traffic.

2.2 Connectivity Enablers in the Transport Network

2.2.1 Architecture

One of the key enabling technologies in the 6Green SBA is related to providing the connectivity between all the elements in the architecture. This means covering network connectivity aspects at cloud and in the transport network. The element responsible for managing the operation of the wide area networks that reach to the cloud is the Wide-Area Infrastructure Manager (WIM).
The WIM oversees orchestrating and managing the transport network infrastructure. It acts as a central entity that controls and configures the connectivity between the different NFVI points of presence in the 6Green SBA architecture. It sets and manages links, routes, and network resources required for the communication between endpoints in the 6Green SBA architecture. The WIM is a key element in managing the connectivity between sites, as it will provide the network slice connectivity at transport level to fulfil the needed requirements of the verticals.

The WIM is involved in the slice realization workflow as follows: upon a slice request, the Network Slice Management Function (NSMF) interacts with the WIM in accordance with the 3GPP TS 28.541 specification [1]. This interaction includes the identification of the network functions involved in delivering the end-to-end slice, as well as the specific slice requirements across the Core, Transport, and RAN domains. Based on this information, the WIM is responsible for establishing the required transport connectivity and for ensuring that the requested Service Level Objectives (SLOs) and Service Level Expectations (SLEs) are fulfilled within the transport domain. The WIM is complemented with a new element called Network Slice Controller (NSC). This new component, defined by IETF [2] is in charge of orchestrating the request, realization and lifecycle control of network slices at transport level. This component translates the abstract slice service requirements to concrete technologies and establishes required connectivity ensuring that resources are allocated to the transport network slice as necessary. It will provide the connectivity in situations such as:

- Extending the connectivity to the cloud and edge by deploying virtual routing and forwarding (VRF) instances. This interaction is complex because many components and resources in the SBA architecture are virtualized on bare metal servers in the cloud. Consequently, communication is abstracted, using virtualized rather than physical interfaces. This requires interaction with the Virtual Infrastructure Manager (VIM), which has the context and knowledge about the mapping of cloud components.
- Providing connectivity and redirection of traffic among the different UPF deployed in the 6Green SBA architecture.
- Providing connectivity in the transport domain upon slice requests involving Decarbonization Level Objectives (DLOs), defined in D4.1. This use case is further developed below.

The NSC comprises two modules: the mapper and the realizer. The mapper processes the customer's request, contextualizing it within the IETF transport network. The realizer then translates this request into a practical implementation of the transport network, fulfilling the slicing request by interacting with the associated network controller within the network.

The request received by the NSC originates from a 6G vertical seeking an end-to-end slice with specific requirements. This request is managed by the 6G end-to-end orchestrator, which configures the RAN and Core Network elements accordingly before passing the request to the NSC for processing. The NSC then feeds the relevant wide area network controllers to implement the network slice within the transport network. The architecture of this component is depicted in Figure 2-6.

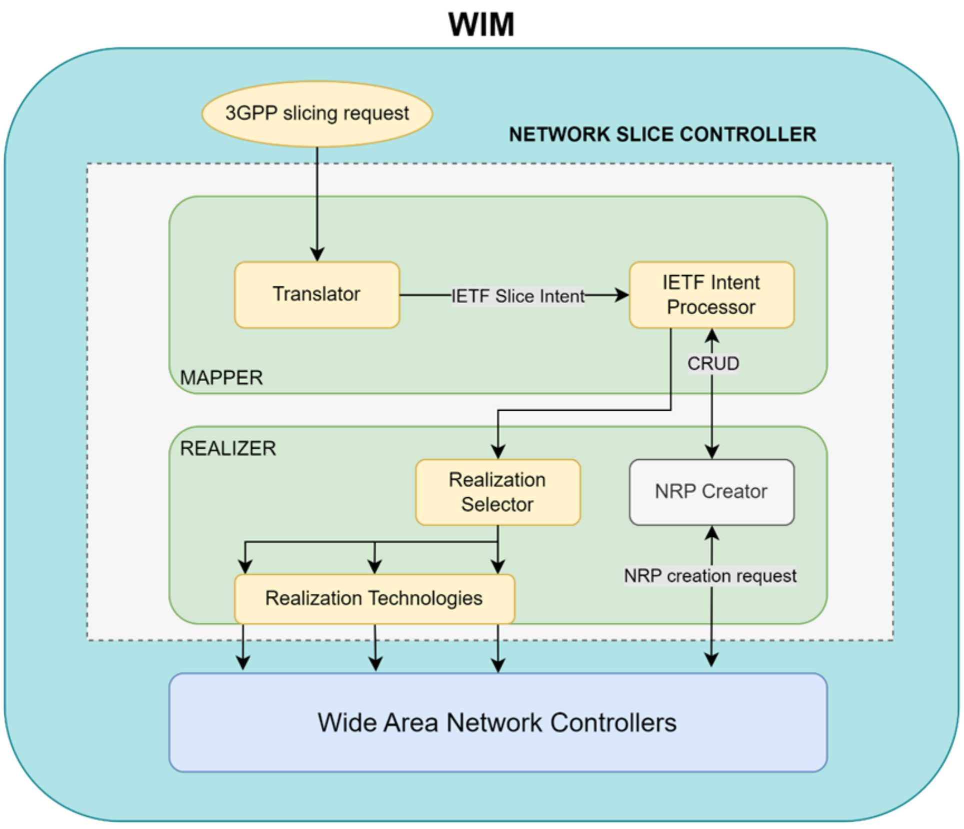


Figure 2-6: WIM Architecture.

The mapper handles client network slice requests and correlates them with existing slices. The workflow is as follows: when a slice request is received, the mapper translates it from 3GPP NRM [1] terms into the IETF NBI data model [3]. This involves identifying the service demarcation points (SDPs) that define the connectivity within the transport network. After identifying and mapping these parameters, the next step is to check the feasibility of implementing the slice request.

To realize a slice, an existing network resource partition (NRP) that meets the specified slice requirements is needed, which may not always be available. Feasibility information is retrieved from an external module, beyond the scope of this definition, which provides a response regarding the feasibility of realizing the slice. If no suitable NRPs are available for instantiating the slice, the mapper requests the realizer to create a new NRP. This involves interacting with the wide-area network controllers responsible for the transport network managed by the NSC. This process is iterative until the mapper determines that the slice realization is feasible.

The realizer module handles the actual implementation of each slice by interacting with specific wide-area network controllers. It receives requests from the mapper and decides on the technologies to use for instantiating the slice based on the selected NRP associated with the slice.

2.2.2 Validation

To evaluate the NSC, it is presented within the context of one of the previously introduced use cases, in which the NSC is responsible for providing a transport network slice that fulfills "green" requirements, also known as Decarbonization Level Agreements (DLAs)—namely, carbon emissions, energy consumption, energy efficiency, and the use of renewable energy sources. It is assumed that the slice request originates from the NSMF component in the form of an intent, following the 3GPP TS 28.541 specification [1].

Additionally, it is assumed that the WAN is managed by the TeraflowSDN controller [4], and that a separate component, referred to as the energy planner, is responsible for computing the most energy-efficient path that satisfies the slicing request requirements.

For the purpose of the evaluation, the planner has been integrated into the NSC. However, as a similar component based on a Path Computation Element (PCE) with energy-aware capabilities has been developed within the 6Green Project by Ericsson, the NSC is fully compatible and capable of interfacing with this alternative solution. The architecture of this use case is presented in Figure 2-7.

The overall workflow is resumed as follows:

1. The NSMF, upon a slice request coming from a vertical, sends the slice intent for the NSC to handle in the transport domain a slice between endpoints A and B.
2. The NSC translate the intent an IETF Slice Service Request [3] and sends it to the planner component.
3. The energy planner component, retrieves energy metrics from Teraflow to perform calculations.
4. With this information, the planner is able to obtain the optimal traffic path that meets the specified requirements in the intent.
5. The planner sends this path to the realizer.
6. The realizer sends a request to TeraflowSDN for creating a layer 2 VPN to realize the slice in the cloud continuum wide area.

Firstly, as mentioned before, the process is triggered by the slice intent coming from the NSMF, following the 3GPP TS 28.541 specification. This specification defines the endpoints of the request, which, for this use case, are A and B, and 4 main slice requirements, defined a Decarbonization Level Agreements (DLAs):

- Energy Consumption, equivalent to the EC indicator, expressed in Joules or kWh [EC].
- Energy Efficiency, equivalent to the EE indicator, expressed in Watts per bits per second [EE].
- Carbon Emissions, expressed in grams of CO2 per kWh [CE].
- Usage of Renewable Energy, expressed as a rate [URE].

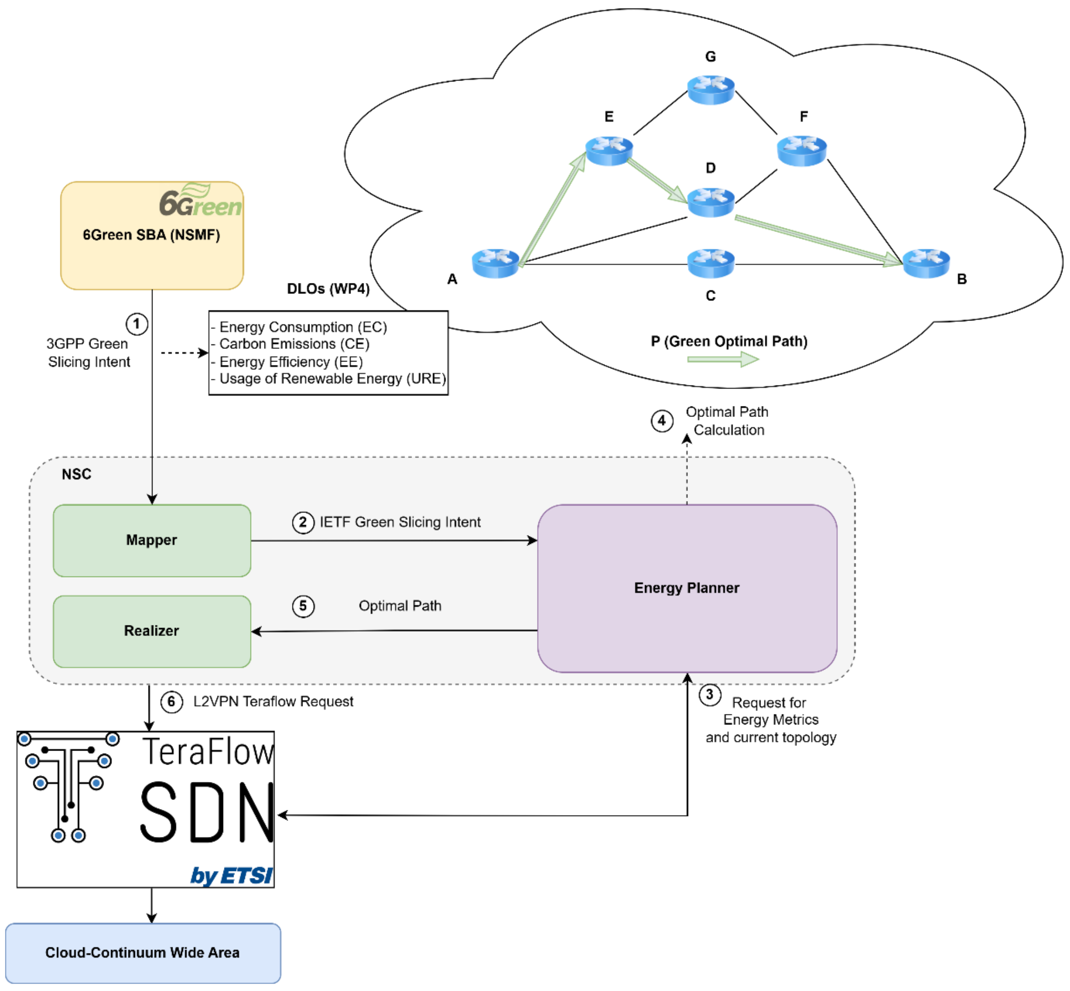


Figure 2-7: Validation Architecture.

For more detail about these parameters, see 6Green D4.1 [5]. An example of a slice service profile with DLAs is presented below. Note that no other SLOs (i.e. latency, throughput) are defined in this use case for simplicity.

```
"CNSliceSubnet": {
 "networkSliceSubnetType": "CN_SLICESUBNET",
 "SliceProfileList": [
  {
   "sliceProfileId": "GREEN_SLICE",
   "CNSliceSubnetProfile": {
    "EnergyEfficiency": 1e-9,
    "EnergyConsumption": 3000,
    "RenewableEnergyUsage": 0.7,
    "CarbonEmissions": 200
   }
  }
 ]
}
```

Then, the mapper component in the NSC processes the request and translates it into an IETF Slice Service request, as shown below.

```
{
 "ietf-network-slice-service:network-slice-services": {
  "slo-sle-templates": {
   "slo-sle-template": [
    {
     "id": "green_template",
     "description": "",
     "slo-policy": {
      "metric-bound": [
       {
        "metric-type": "energy_consumption",
        "metric-unit": "Joules",
        "bound": 3000
       },
       {
        "metric-type": "energy_efficiency",
        "metric-unit": "GigaWats/bps",
        "bound": 1
       },
       {
        "metric-type": "carbon_emission",
        "metric-unit": "grams of CO2 per kWh",
        "bound": 200
       },
       {
        "metric-type": "renewable_energy_usage",
        "metric-unit": "rate",
        "bound": 0.7
       }
      ]
     }
....
```

After that, once the IETF request is generated, it is sent to the energy planner component. There are two possible deployment options for this component.

- Operation as an external element: interaction with the energy-aware PCE developed in the context of the 6Green Project. This requires doing a request to /sss/v1/slice/compute endpoint API including as body a SliceInput object, which contains de following parameters:
  - requestId: sequential id of the request
  - clientName: the identifier of the client issuing the request
  - graph: the descriptor of a service graph, which includes the nodes in which the slice is deployed (e.g. A and B), the logical link between the nodes implementing the slice topology and the slice constraints, mapped to the DLOs described above.

The PCE answers with a SliceInfo object that includes, among other parameters, the path computed for the requested slice.

- Rely on an internal planner integrated within the NSC: responsible for computing the most energy-efficient path.

For the purposes of the isolated evaluation of this use case, the internal planner embedded in the NSC has been chosen for evaluation. Therefore, the description from this point corresponds to the use of this internal planner.

This way, the energy planner retrieves the energy metrics from the nodes in the topology by requesting TeraflowSDN to obtain these metrics from all nodes. These are:

- Power consumed by each node in idle state, measured in Watts [P_{idle}]
- Power consumed by components in nodes (e.g. transceivers, boards), measured in Watts [$P_{components}$]
- Energy efficiency of each node, measured in Watts per bit per second [ee]
- Usage of renewable energy, measured as a rate. This data is specific to the plant where the node is located [ure]
- Carbon emissions, measured in grams of CO2 per kWh. This data is specific to the plant where the node is located [ce]

These values are obtained from the TFS Analytics component, which processes the instantaneous energy metrics obtained from nodes in the TFS Energy Collector component. See Figure 2-8 for TFS component architecture. Traffic through the nodes is assumed to be 100 gbps and the measurement time window is assumed to be one hour long.

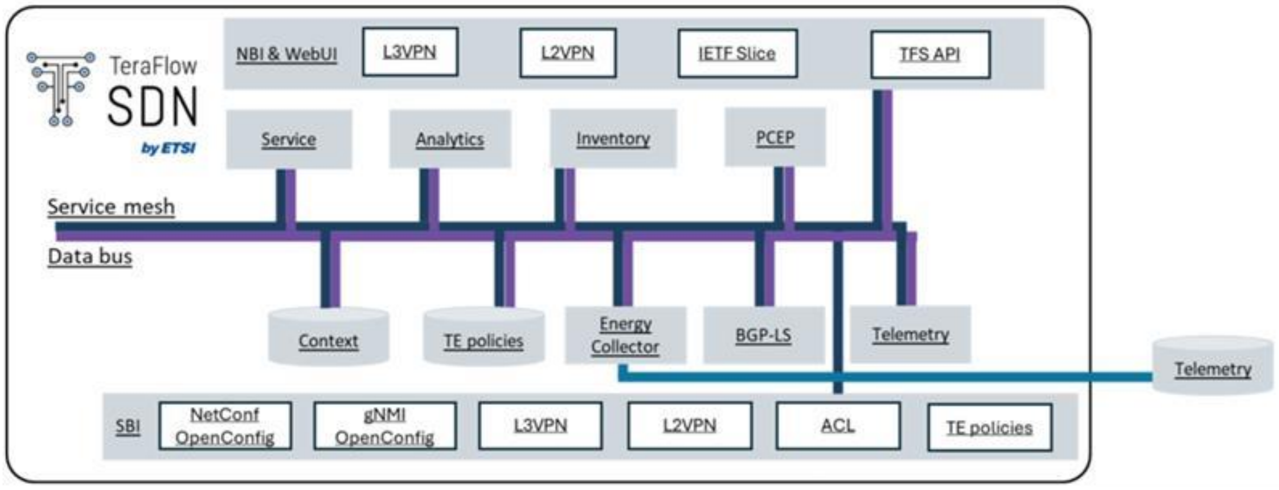


Figure 2-8: TFS Components Architecture.

The YANG model defining the energy metrics collected from nodes is depicted below:

```
module: static-device-energy-tid
 +-- container device
   +-- leaf name              string
   +-- leaf typical-power       decimal64 (W)
   +-- leaf maximum-traffic     decimal64 (Gbps)
   +-- leaf max-power           decimal64 (W)
   +-- leaf efficiency          decimal64 (W/fps)
   +-- leaf nominal-power       decimal64 (W)
   +-- leaf carbon-emissions    decimal64 (gCO2/kWh)
   +-- leaf renewable-energy-usage decimal64 (rate)
   +-- list power-supply
     +-- key "name"
     +-- leaf name            string
     +-- leaf type           string
     +-- leaf capacity        decimal64 (W)
     +-- leaf typical-power    decimal64 (W)
     +-- leaf nominal-power    decimal64 (W)
   +-- list boards
     +-- key "name"
     +-- leaf name            string
     +-- leaf type           string
     +-- leaf capacity        decimal64 (W)
     +-- leaf typical-power    decimal64 (W)
     +-- leaf nominal-power    decimal64 (W)
   +-- list components
     +-- key "name"
     +-- leaf name            string
     +-- leaf type           string
     +-- leaf capacity        decimal64 (W)
     +-- leaf typical-power    decimal64 (W)
     +-- leaf nominal-power    decimal64 (W)
   +-- list transceivers
     +-- key "name"
     +-- leaf name            string
     +-- leaf type           string
     +-- leaf capacity        decimal64 (W)
     +-- leaf typical-power    decimal64 (W)
     +-- leaf nominal-power    decimal64 (W)
```

Taking into account the topology in the wide area network (Figure 2-9), the energy planner builds a weighted graph of the topology (Table 1). The weight of each node is named as *Green Index* [GI], measured in grams of CO2, and it defines how “Green” is each node in the topology. The planner computes the shortest path by applying a Dijstra Algorithm [6]. The formula that the planner uses to compute the nodes’ GI is depicted below:

$$\boldsymbol{GI} = \left(\mathrm{P}_{idle} + \mathrm{P}_{components} + ee \times traffic\right) \times \frac{time}{1000} \times (1 - ure) \times ce$$

where Pidle, Pcomponents and ee*traffic is calculated based on the studies in [7].

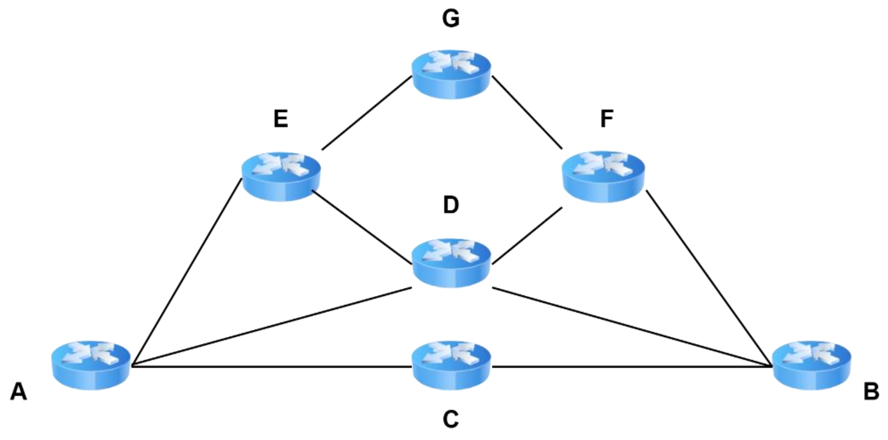


Figure 2-9: Wide Area Network Topology.

	A	B	C	D	E	F	G
GI	145	450	282	380	355	242	226

Table 1: Weighted Adjacency Matrix.

Subsequently, the green optimal path is computed as follows:

$$P_{opt} = \min_{P \in \mathcal{P}(A,B)} \sum_{v \in P} GI(v)$$

being *v* a node of the set

$$\boldsymbol{P} = \{\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}, \boldsymbol{E}, \boldsymbol{F}, \boldsymbol{G}\}$$

with the following restrictions taking into account the slice requirements

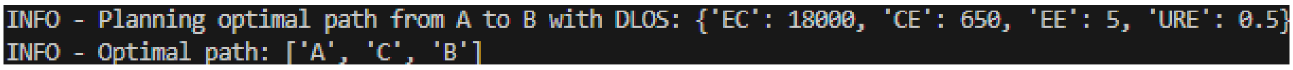
$$\sum_{v \in P} ee(v) \leq EE$$

$$\forall v \in P,\ ure(v) \geq URE$$

$$\sum_{v \in P} ce(v) \leq CE$$

$$\sum_{v \in P} ec(v) \leq EC$$

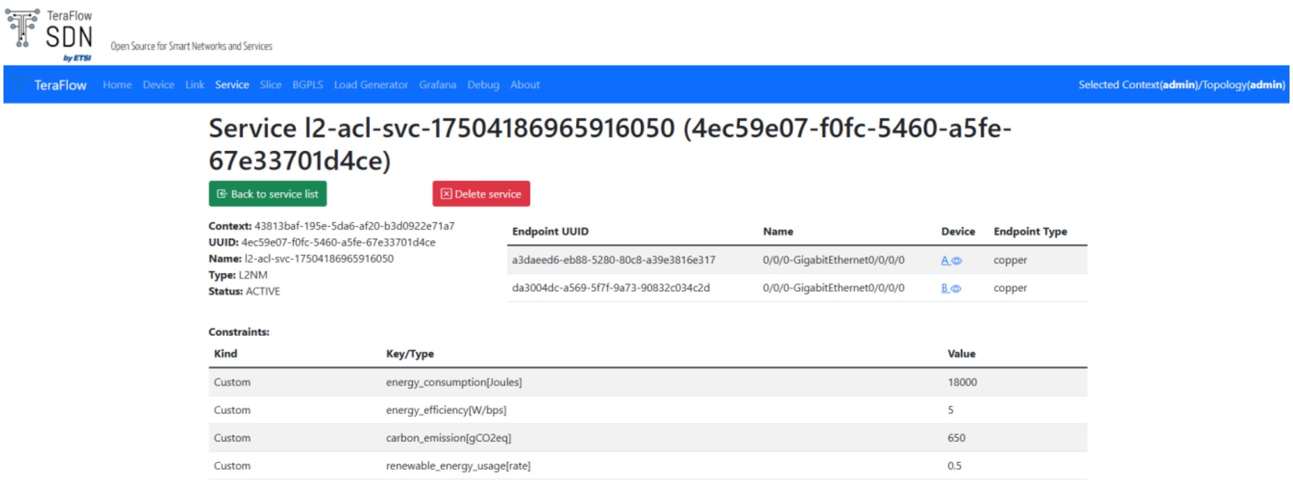
After the request is sent to the planner, it responds with the following path between A and B (Figure 2-10).



```
INFO - Planning optimal path from A to B with DLOS: {'EC': 18000, 'CE': 650, 'EE': 5, 'URE': 0.5}
INFO - Optimal path: ['A', 'C', 'B']
```

Figure 2-10: Optimal Path calculated by the planner.

Ultimately, the NSC generates a L2VPN TeraflowSDN service template between the specified endpoints over the optimal path defined by the energy planner and loads it into TeraflowSDN. If we access TeraflowSDN, the service is deployed, as shown in Figure 2-11, with the constraints defined in the request. Furthermore, the specific configurations and the path for traffic is depicted in Figure 2-12.



Service l2-acl-svc-17504186965916050 (4ec59e07-f0fc-5460-a5fe-67e33701d4ce)

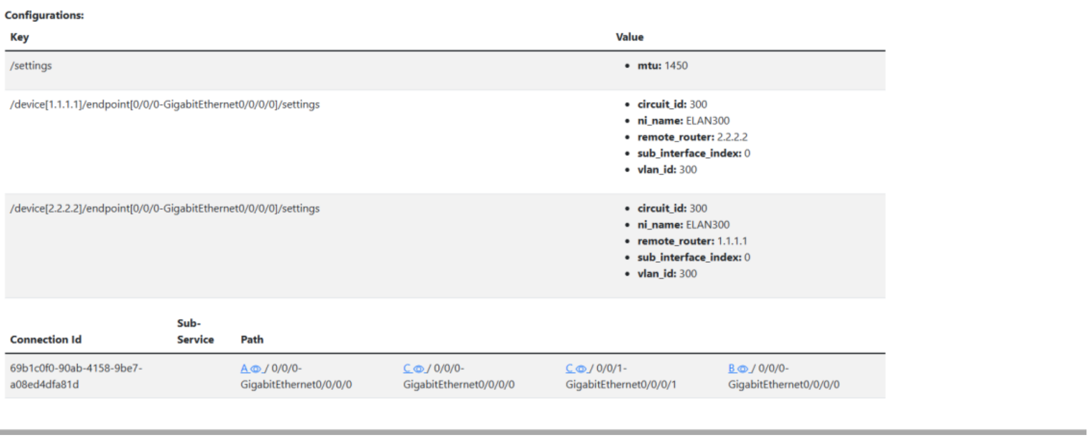
Context: 43813baf-195e-5da6-af20-b3d0922e71a7
UUID: 4ec59e07-f0fc-5460-a5fe-67e33701d4ce
Name: l2-acl-svc-17504186965916050
Type: L2NM
Status: ACTIVE

Endpoint UUID	Name	Device	Endpoint Type
a3daeed6-eb88-5280-80c8-a39e3816e317	0/0/0-GigabitEthernet0/0/0/0	A	copper
da3004dc-a569-5f7f-9a73-90832c034c2d	0/0/0-GigabitEthernet0/0/0/0	B	copper

Constraints:

Kind	Key/Type	Value
Custom	energy_consumption[Joules]	18000
Custom	energy_efficiency[W/bps]	5
Custom	carbon_emission[gCO2eq]	650
Custom	renewable_energy_usage[rate]	0.5

Figure 2-11. TFS L2 service created



Configurations:

Key	Value
/settings	• **mtu:** 1450
/device[1.1.1.1]/endpoint[0/0/0-GigabitEthernet0/0/0/0]/settings	• **circuit_id:** 300 • **ni_name:** ELAN300 • **remote_router:** 2.2.2.2 • **sub_interface_index:** 0 • **vlan_id:** 300
/device[2.2.2.2]/endpoint[0/0/0-GigabitEthernet0/0/0/0]/settings	• **circuit_id:** 300 • **ni_name:** ELAN300 • **remote_router:** 1.1.1.1 • **sub_interface_index:** 0 • **vlan_id:** 300

Connection Id	Sub-Service	Path			
69b1c0f0-90ab-4158-9be7-a08ed4dfa81d		A / 0/0/0-GigabitEthernet0/0/0/0	C / 0/0/0-GigabitEthernet0/0/0/0	C / 0/0/1-GigabitEthernet0/0/0/1	B / 0/0/0-GigabitEthernet0/0/0/0

Figure 2-12: Configurations and path for service.

3 Cloud Native Orchestration and Automated Network Infrastructure Management

3.1 Deployment Aspects

Cloud Native Orchestration and Automated Network Infrastructure Management for Green Efficiency in 6G

In 6Green, Cloud-native technologies are essential for the efficient operation and optimization of our complex 5/6G systems, enabling end-to-end (E2E) management and automation services. It is known that 6G networks aim to enhance wireless network capabilities, delivering higher data rates, lower latency, and massive connectivity for diverse applications and services. This involves seamless management of network resources from end devices to core networks, utilizing advanced automation and AI techniques to optimize performance and efficiency.

The AI/ML is leveraged for predictive analytics, network slicing management, and real-time decision-making, enhancing automation capabilities. Cloud-native capabilities, enabled through Service-Based Architecture (SBA), improve modularity and flexibility in network functions and services, based on the following factors:

- **Microservices Architecture**: Facilitates independent development, deployment, and scaling of applications, enhancing agility and resilience.
- **Containerization**: Encapsulates services within containers, ensuring consistency across environments and simplifying deployment and scaling.
- **Orchestration (Kubernetes)**: Manages the lifecycle of containers, automating deployment, scaling, and management of containerized applications.
- **DevOps Practices**: Integrates development and operations teams to improve collaboration and accelerate service delivery.
- **Continuous Integration/Continuous Deployment (CI/CD)**: Automates the software delivery process, enabling frequent and reliable updates.
- **Service Mesh**: Manages service-to-service communication, providing load balancing, service discovery, and secure connectivity.

In 6Green, we deliver Cloud Native with Green Efficiency supported by Cloud Infrastructure, the Automation functions across RAN, Core, and Transport systems, streamlining processes and enhancing performance (6G RAN, 6G Core, and 6G-EDGE). The Dynamic optimization of the resource utilization is also implemented, ensuring consistent performance and energy efficiency, addressing key metrics such as Quality of Service (QoS). The general approach is to integrate NF Sets to ensure 6G Service-Based Interfaces (SBIs) interoperability and flexibility, enabling tailored solutions that meet specific needs and adapt to evolving technologies and demands. This delivers transformative, flexible consumption of Network Services, providing a scalable, flexible, and cost-effective way to manage network infrastructure.

The principles rely on virtualized network functions (VNFs) that abstract traditional networking functionalities into software-based components, with Microservices architecture for modularity, scalability, and agility. The cloud native invokes the containerization and orchestration, which automate the lifecycle management of networking containers, ensuring scalability, resilience, and efficient resource utilization. API-Driven mechanisms are enabled for programmable and automated network management and provisioning. As implementation of dynamicity and scalability, 6Green allows network resources to scale up or down in response to changing demand, enabling efficient resource allocation, cost optimization, and improved performance of network services. The entire ecosystem is based on monitoring and observability mechanisms, the tools which provide visibility into network performance, health, supporting the AI/ML data framework and further the prediction.

6Green empowers the full 6G potential by integrating the sets of NFs and data analytics processes (NSDAF, NWDAF) through interoperable and APIs. This demonstrates the envisioned service-based architecture and orchestration that is applied for optimized applications placement and execution in 6G environments, automated cloud/edge-computing scaling of applications, dynamic creation and maintenance of optimized services, and automated network infra control of facilities through Infrastructure as Code. 6Green drives energy efficiency, agility, and innovation in operations.

Cloud Native Orchestration and Automated Network Infrastructure Management for Green Efficiency in 6G example: NSDAF Use Case.

The NSDAF (Network Slice Data Analytics Function) module has been developed within the 6Green Project, focusing on energy consumption prediction for network slices. The NSDAF aims to infer KPIs, estimate energy consumption (and carbon footprint) of network slices, including edge-cloud resources hosting vertical applications, and analyze infrastructure data and network slice metrics.

Data Flow and Architecture:

The NSDAF collects data from various sources:

- Redis DB/Channels: Receives power measurement data for containers and machines, identified by a unique slice_id.
- Prometheus: Collects infrastructure data (CPU/RAM usage).
- NWDAF/MDAF: Receives KPIs.

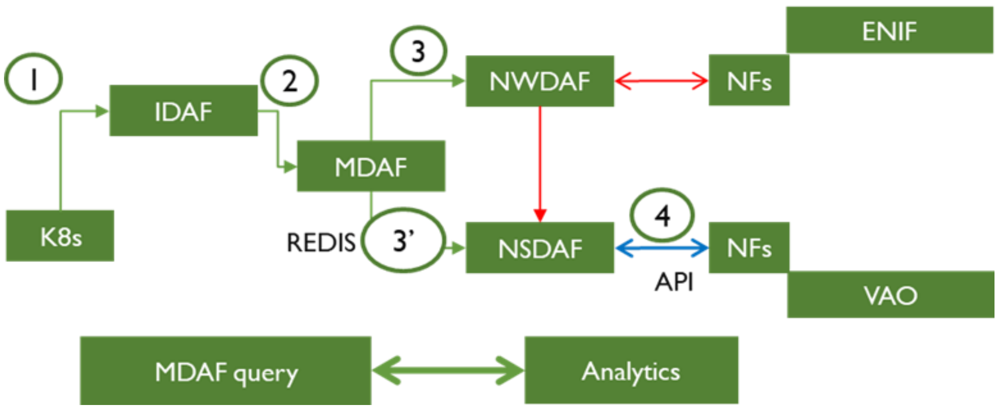


Figure 3-1: NSDAF Interactions with other functions.

The collected data is processed and stored in Redis, and a Flask web service exposes REST APIs for retrieving historical data and future predictions. AI/ML algorithms, specifically ARIMA, are used to forecast energy consumption.

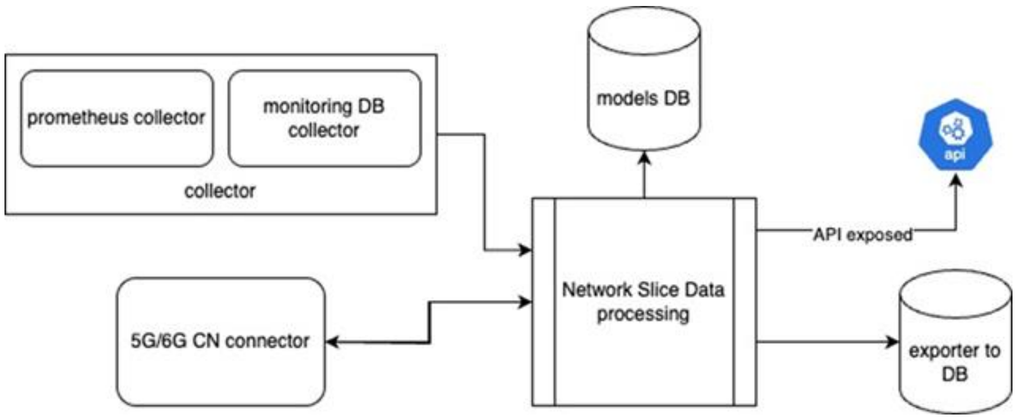


Figure 3-2: NSDAF data processing.

Specifically, the NSDAF utilizes the ARIMA model for time series energy forecasting, ARIMA models are temporal dependencies in the data, and the model components are:

- AR (p): Models the relationship with past observations.
- I (d): Applies differencing to make the data stationary.
- MA (q): Models the relationship with past forecast errors.

The ARIMA parameters (p, d, q) can be automatically tuned or set statically.

As described in Figure 3-3, the NSDAF data flows is based on:

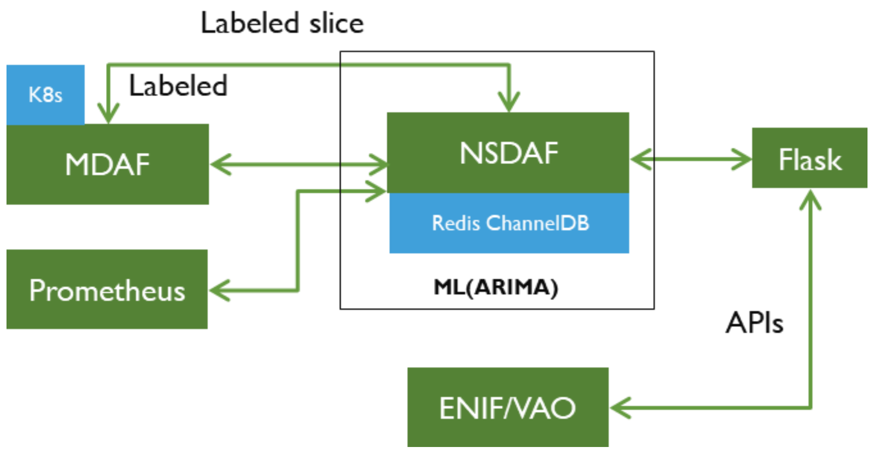


Figure 3-3: NSDAF data flows.

- NSDAF collect energy consumption data, aggregates it daily and leverages a prediction model to forecast future consumption.
- Flask web service with two main endpoints - one for retrieving current historical data and one for obtaining future predictions.
- Different ML models add flexibility to ensure the model can adapt to the generated data characteristics.

- ENIF/ VAO and other systems to easily obtain both historical insights and future predictions through simple REST API calls.
- NSDAF lead to additional analyses - carbon footprint estimation - applying conversion factors to the energy data.
- This makes the tool highly valuable for energy monitoring, forecasting, and environmental impact assessments.

Both actual and predicted energy consumption power measurements are provided through APIs, data is processed to calculate the daily average energy consumption per slice_id. The Data is returned as a list of tuples: date string energy value, in float List of available slice_ids should be retrieved. ML algorithms build forecasts for consumption of components in network slice / level of load or resource usage on specific server and power consumption related to it.

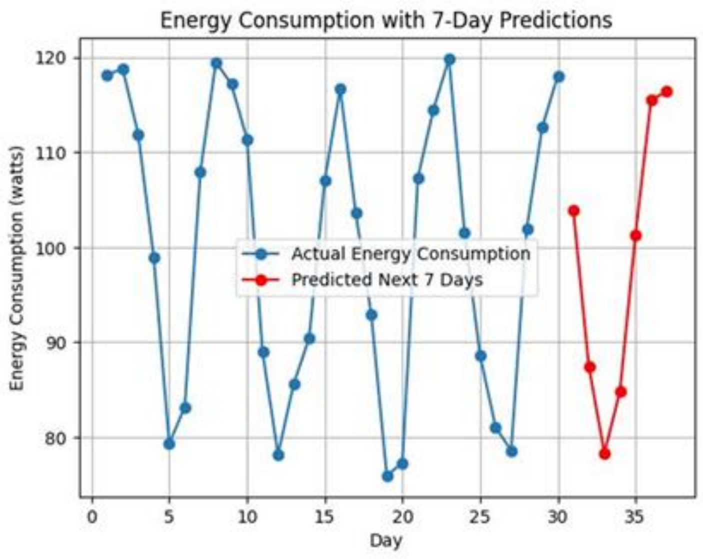
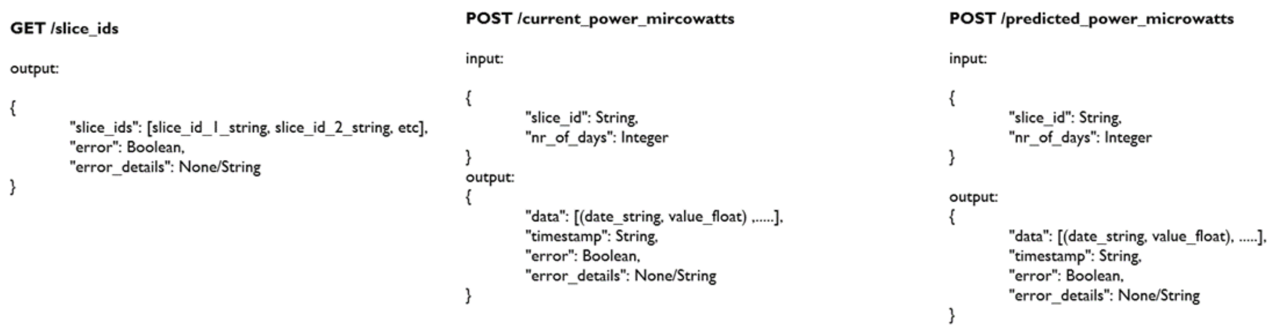


Figure 3-4: Actual and predicted energy consumption by NSDAF

As described previously, in relation with 6Green (SBIs and APIs) and NFs interworking, the NSDAF delivers the following outputs:

- GET /slice_ids Endpoint: returns a list with all slice ids available in order to return energy metrics per slice ids
- POST /current_power_mircowatts Endpoint responds to POST requests by returning historical energy consumption data
- POST /predicted_power_microwatts Endpoint forecasting future energy consumption
- POST /cpu_usage Endpoint extracting CPU usage information (similar RAM/Disk/Net)

An example of cloud native Orchestration and Automated Network Infrastructure Management for Green Efficiency, APIs implementation of NF level is provided in Figure 3-5.



```
GET /slice_ids

output:

{
        "slice_ids": [slice_id_1_string, slice_id_2_string, etc],
        "error": Boolean,
        "error_details": None/String
}
```

```
POST /current_power_mircowatts

input:

{
        "slice_id": String,
        "nr_of_days": Integer
}
output:
{
        "data": [(date_string, value_float) ,.....],
        "timestamp": String,
        "error": Boolean,
        "error_details": None/String
}
```

```
POST /predicted_power_microwatts

input:

{
        "slice_id": String,
        "nr_of_days": Integer
}

output:
{
        "data": [(date_string, value_float), .....],
        "timestamp": String,
        "error": Boolean,
        "error_details": None/String
}
```

Figure 3-5: NSDAF APIs

The 6G networks bring the promise of network enhanced performance, with an increased focus on sustainability and energy efficiency. The optimizing energy consumption is possible within cloud-native environments, in this particular case for multi-tenant deployments. The tenant evaluation explores in this case the integration of tenants within a cloud-native architecture. At this stage, the tenant integration is focusing on the achievements results of Network Function Set 1 (NF SET 1) and the evaluation of energy-related Service Level Indicators (SLIs) and Key Performance Indicators (KPIs).

The energy analytics is treated as a first-class, within the multi-tenant context concern, as tenants are isolated at runtime (via Kubernetes namespaces/pods) and the relevant resource (containers, VMs, servers) are labelled with a tenant identifier (slice_id). This labeling lets NSDAF correlate infrastructure metrics (CPU, RAM, disk, network) with power data on a per-tenant basis, enabling accurate attribution of energy use and impact. It defines and reports tenant-specific energy KPIs (consumption per unit, reduction, efficiency) and supports tenant-level reports and dashboards. With respect to this, forecasts are produced for each tenant (rather than only at slice/system level), and daily aggregation is aligned to tenant identifiers so that both historical and predicted series reflect tenant behavior. The same REST interfaces described in Section 3.1 are reused, but the semantics are explicitly per tenant (e.g., /slice_ids enumerate active tenant IDs). An additional topic is the closed-loop, tenant-aware orchestration, as the NSDAF integrates with ENIF and the Virtualized Automation Orchestrator (VAO), so energy insights can trigger tenant actions (for example, pre-emptive scaling or resource rightsizing) with policy control at tenant granularity. In a larger context, the operators can set energy-aware policies per tenant (e.g., caps, throttles, or prioritization) to support sustainability targets without imposing uniform rules across all workloads.

3.2 Service Mesh Technologies

With the rising popularity of cloud-native applications, service mesh architectures have emerged to enable advanced functionalities within PaaS environments. Service meshes rely on sidecars, that are "accompanying" containers, in particular they allow interacting with other containers by communicating with their respective sidecars. In a nutshell, their main features are *i*) connectivity, including service discovery, *ii*) monitoring, through tools such as Prometheus, and iii) security, with ad-hoc policies to manage accesses.

In the context of 6Green, the service mesh paradigm has been investigated to enable the project's innovations on Kubernetes clusters, with special focus on Edge Agility in this phase of the project. Edge Agility is meant to provide smart, fast, and automated horizontal scalability to vertical application and related slices across the 5/6G edge-cloud continuum, for example in reaction to a handover event or to move the workload where more convenient (e.g., to consolidate vApps and slices or to exploit the presence of renewable sources). In this respect, the first step of interest is the so-called "scale to zero", that is switching off the pods for a certain non-used service for zeroing its, and to quickly resume the operating capacity when needed. One of the most intuitive ideas is that we can try to pause what is not used, like when we turn off the light we don't need it.

The main issues to be investigated are the following:

- What happens when some traffic arrives at a service that has been scaled to zero?
- How much time is needed to resume the service? (Scale to zero)

A partial solution to both problems could be the traffic steering to another support cluster/zone where the service is online, but this requires having a multi cluster environment. In this case we would have all the time we need for the service to be resumed, but a compromise on the QoS may have to be made (more latency for the original zone, and more load on the support zone). When the service in the original zone comes up again, we can steer the traffic back where it should go. A bigger problem arises when dealing with continuous connections, is it possible to change the destination of the traffic without a service drop? It could be possible, but this case has to be managed by the application/service.

Another interesting question is when do we need to restore the service? For the sake of simplicity, let's limit ourselves to the HTTP case. If a request arrives at the service, is it a sufficient reason to restart the service? For essentials services we could think of delegating the answer to a support cluster (see traffic steering) while, for not essential services, it may depend on the service provider policy. In any case, the operation of turning on/off the service multiple time could lead to power consumptions that are higher than the case where the service is always online. This is why, in some cases, we could think not to respond to a request.

To manage scale to zero operations in this context, the Kubernetes scaling mechanism is not enough since domain-specific knowledge may be required. To solve this problem, additional tools called Operators can be installed on the cluster. Operators are software extensions to Kubernetes that manage applications and their components using custom resources that allow to define application-specific controllers for complex applications. Controllers are the Kubernetes components that manage the resources lifecycle to bring the cluster state closer to the desired one.

In the context of 6Green, the operators that we are considering are Knative and KEDA. Knative[1] provides a common toolkit and API framework for serverless workloads, to support the deployment, running, and management of serverless, cloud-native applications to Kubernetes, but it does not allow to be integrated on an already running cluster. KEDA[2] is a Kubernetes-based Event Driven Autoscaler[3]. It allows the scaling of any container in Kubernetes based on various metrics like the number of events needing to be processed. It is lightweight and can work with standard K8s components such as Horizontal Pod Autoscaler and can extend functionality without overwriting or duplication. While it natively offers the needed scaling feature, it can only operate on a per-cluster basis, which would prevent us from scaling in the continuum. When a scale to zero operation is performed on essential services, the traffic needs to be redirected to a running instance of the service in another cluster to prevent service interruption. For this purpose, a service mesh should be installed on the cluster. In 6Green, we decided to rely on Istio as a service mesh, that is described in the next subsection, followed by a set of evaluation results.

Alongside the traditional Kubernetes networking we have decided to use Istio. This enables us to extend Kubernetes establishing a programmable, application-aware networking using the Istio provided Envoy

[1] https://www.redhat.com/en/topics/microservices/what-is-knative
[2] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-custom-metrics
[3] https://dev.to/sarony11/hpa-vs-keda-in-kubernetes-the-autoscaling-guide-to-know-when-and-where-to-use-them-m96

service proxy. Beyond this, it allows us to use Monitoring data generated by the sidecars (Envoy proxy) to energy efficiency scopes (as explained later in the document).

We documented how Istio is working to understand when it is possible to install it, in particular if it possible to perform a Day-2 installation on a cluster with services that are already running on it.

First of all, the installation is performed using the dedicated Istioctl CLI tool or through the Helm chart. After this task, the injection of the sidecar must be enabled for every namespace that is needed. Now, if the cluster is brand new, we do not need to worry about anything, Istio will be working on deployed services. Otherwise, if some services are already running on the cluster, we will need to change these resources. To understand why, Istio uses Admission Controllers to intercept APIs calls and in this process, it injects sidecars into running Pods:

- Prior to persistence of Resources
- After the API request has been authenticated and authorized

Scale-to-zero has been implemented by means of a UPF prototype, based on the Berkeley Extensible Software Switch (BESS[4]): when the UPF receives a packet directed towards the application deployed on the "scaled-to-zero" pod, it sends an alert to the pod asking it to scale back up and, in the meantime, it stores the incoming to give it time to get back up without packet losses. Although the mechanism itself is straightforward, on the other hand the operation can introduce additional consumptions if the scaling operation happens too often. The following results allow for an evaluation of such consumption for a better understanding of how to best apply the mechanism.

3.2.1 Overhead of Container Scaling Operations

Studies have been done to see the advantages in scaling pods to zero when needed by observing the CPU power consumption. These data are relevant in understanding when it is applicable to horizontally scale vertical application and related slices across the edge-cloud continuum. In particular, several measurements and results that have been performed on the CPU power consumption ascribable to the pod deployed on the Kubernetes Cluster when it scales from zero to one and vice versa with different timings using Intel "Running Average Power Limit" (RAPL)[5][8].

[4] https://span.cs.berkeley.edu/bess.html
[5] Intel RAPL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html

3.2.1.1 Setup

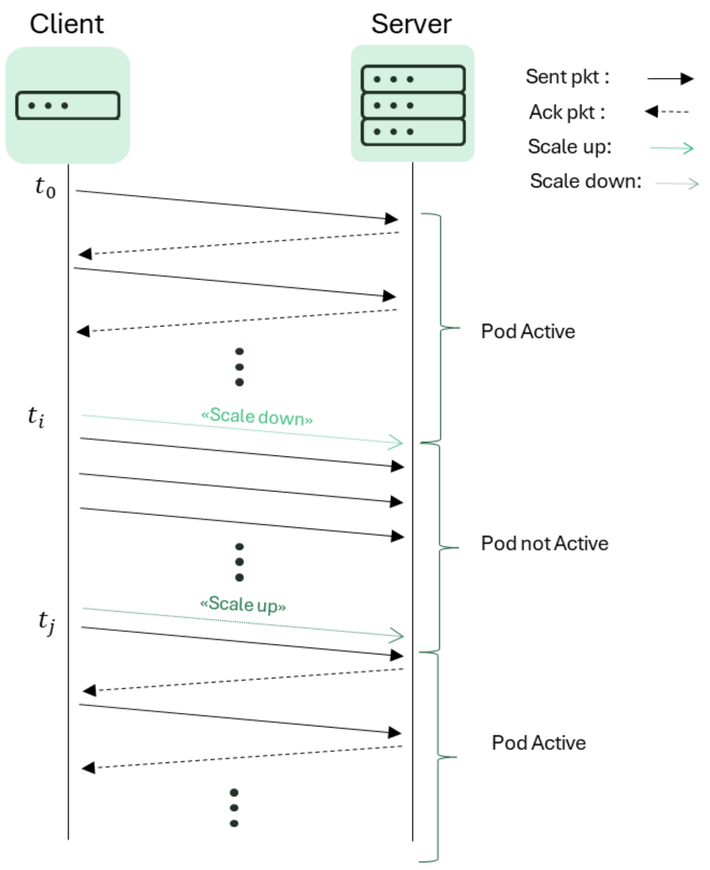


Figure 3-6: Traffic generation and scaling the pod.

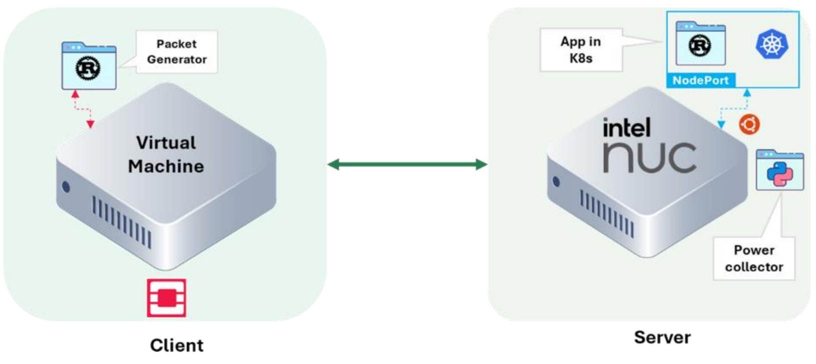


Figure 3-7: Setup used for the tests.

To simulate a server, a NUC PC has been used. This PC was developed by Intel, and it can be used in both gaming and commercial fields. The mounted processor is an Intel(R) Core (TM) i7-6770HQ CPU @ 2.60GHz. On top of it, a Kubernetes cluster has been set up on bare metal, with a single node running both as master and worker, as this configuration better suits the NUC architecture. The operating system installed on it is Ubuntu 24.04.1 LTS.

On the other side, to simulate the client, a virtual machine has been instanced using OpenStack[6] , which is an open source set of software components that is used for cloud IaaS, facilitating the control large pools of networking, the computation and storage of resources and allows to set up the test environment.

In order to assess the CPU power consumption ascribable to the scaling pod, a Python application called Power Collector[7] has been used in the NUC. This application reads the RAPL power counter contained in the Linux kernel throughout the duration of a test every 1,55 seconds, and then saves them. The pod is deployed on the NUC using a Rust application[8] and requires a NodePort, a Kubernetes service that exposes the application onto an external IP address to allow the cluster to communicate with the external world.

On the client side, a self-developed Rust application called Packet Generator[9] that generates continuous traffic to be sent to the pod deployed on the server by opening a TCP/UDP socket. This application sends a packet at $t0$ to the pod deployed on the server. If the pod is active, it elaborates the packet receive while doing some background work and sends an acknowledgement to the client. This goes on until, at time ti, a "scale-to-zero" signal is requested by the client. The server scales the pod to zero by deleting the deployment. All packets sent by the client from here are not elaborated by the server, so they reach a timeout and are then dropped. This mechanism works as a sort of ping, which is used to monitor if the pod is active or not. This goes on until at time tj a "scale-to-one" signal is sent by the client. The server reactivates the pod by deploying a new Kubernetes deployment. These operations go on until the end of the test time. This mechanism can be seen in Figure 3-6.

The Packet Generator application allows the selection of cores to be used in the server during the test, the kind of background operations it does whether a packet is received or not, and the percentage amount of background work. It also allows to choose how the packets are handled by the server, such as the packet elaboration time and the packet elaboration work type, as well as the maximum response time a sent packet and whether the traffic is TCP/UDP. Furthermore, it permits to set the test duration and when a pod should be scaled to zero or to one.

The setup adopted for the test is shown in Figure 3-7. The Virtual Machine is exposed to the internal network by OpenStack by assigning a floating IP to the instance. The IP address assigned to the virtual instance is 192.168.254.166. As for the NUC, the IP assigned to it is 192.168.17.149. As for the Packet Generator, the type of operation used by the background work and the packet elaboration type is the calculation of prime numbers. The number of cores used are 8, which is the total amount of cores the NUC has. The elaboration time of a packet is 1500ms and the max response time of a sent package is 1550ms.

The packets are sent every 1.5 seconds, and test have been conducted by scaling the pod every 15 seconds, 3 minutes, 5 minutes and 12 minutes, and they are compared to when the pod is active with traffic received, the pod is active while in idle and the pod active with traffic dropped. Each test runs for one hour and a few minutes and is repeated for each event.

3.2.1.2 Results Evaluation

As mentioned before, results will focus on power consumption. Figure 3-8 to Figure 3-14 show the power consumption of the server that has been retrieved by the RAPL power counter. Figure 3-8 shows the power consumption of the pod that stays active and in idle state, without any traffic being received. Figure 3-9 to Figure 3-12 show the power consumption when the pod scales with different frequencies, and Figure 3-13

[6] Openstack https://www.openstack.org/

[7] Power Collector https://github.com/nikyjane15/Power_Collector

[8] Rust https://www.rust-lang.org/

[9] Packet Generator https://github.com/s2n-cnit/pktgen

shows the pod always active and receiving traffic, while Figure 3-14 shows the active pod receiving traffic and at certain point it does not receive it anymore.

Focusing on the plots, Figure 3-9 to Figure 3-12, when the pods scales from zero to one there is a spike of power, which is due to the “waking up” of a new pod upon a scaling up request. This power spike is the same height in all the plots. This is different from the case shown in Figure 3-8 and Figure 3-13; in these, the power consumption is flatter. Deploying a new pod means allocating resources and consuming energy in order for it to do its task. After the spike, the power returns to regime until the next scaling request.

Moreover, when observing Figure 3-14, it can be seen that at 00:30:22 the pod doesn’t receive any traffic from the client, and the power consumption from that point forward is 8W, the same mean consumption as Figure 3-8. Comparing it with Figure 3-9 to Figure 3-12, it can be observed that, when the “scale-to-zero” operation is requested, the power consumption while the pod is down is 4W. This means that scaling pods down halves the amount of power consumption with respect to leaving the pod up, independently of whether traffic is transmitted or not.

Another thing worth mentioning is that, by comparing the plots, the power consumption seems to be less when the pod is active all the time and not receiving traffic, and not when it gets scaled to zero. But when the pod is active and receives a continuous stream of traffic, the power consumption doubles with respect to scaling. This can be shown clearly in Table 2 by looking at the mean value in each case. The mean power consumption of the active pod when it just exists is half with respect to scaling and a third of when the pod is active and receiving traffic. Furthermore, the power consumption is the same independently of the total amount of times the pods scales. Looking at the standard deviation, it is worth mentioning that it is much larger when the scaling is happening due to spikes given by the “wakeups” of the pod.

These results show a supposed advantage in scaling the pod to zero to decrease the overall power consumption, but this also depends on the application deployed on the pod. If an application has more overhead when starting up, this could cause a long burst of power consumption before going to regime instead of a spike of power, which means higher consumption.

Table 2: Mean and standard deviation of the power in watts when scaling with different frequencies.

	Mean	Std
Pod active in idle	8	1.28
15 seconds scaling	14	9.79
3 minute scaling	13	9.79
5 minute scaling	14	10.94
12 minute scaling	14	9.82
Pod active with traffic	23	0.1
Pod active with traffic stopped	16	7.65

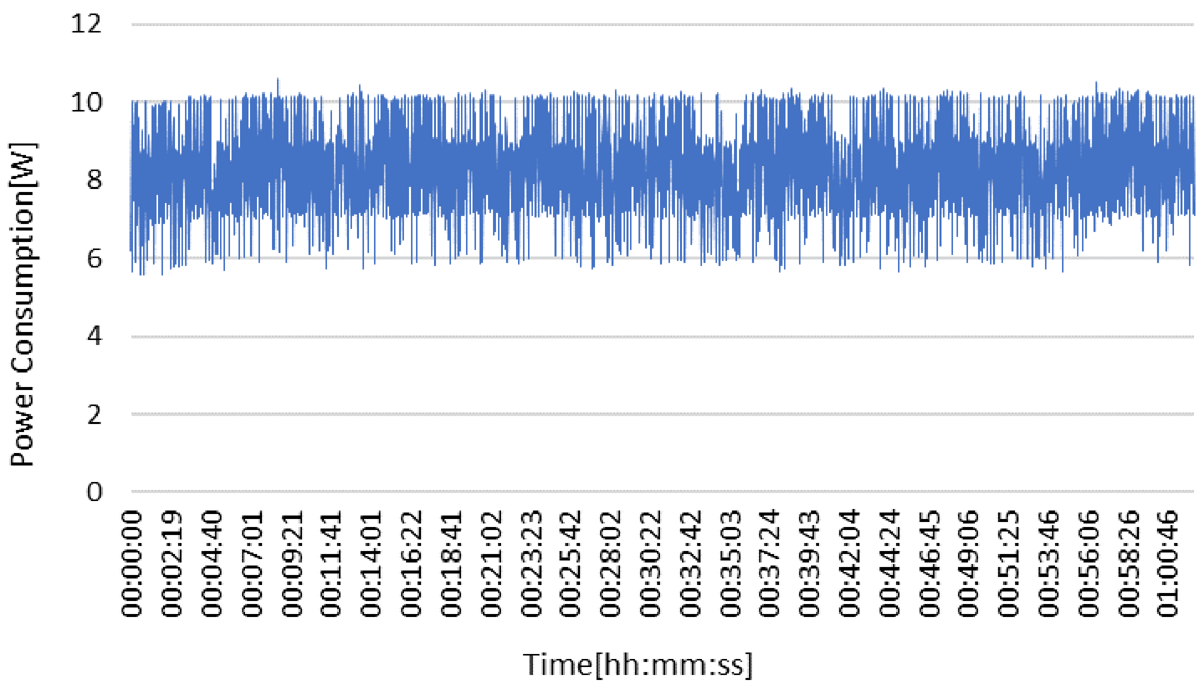


Figure 3-8: Power consumption when the pod does not receive any traffic.

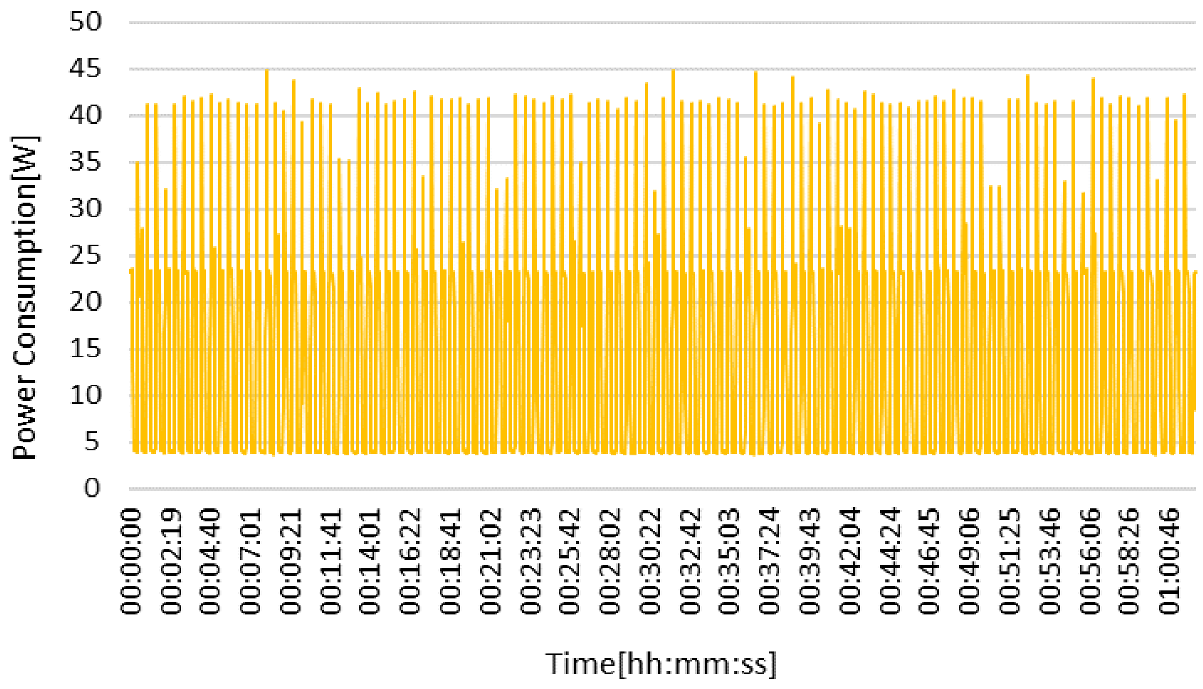


Figure 3-9: Power Consumption when the pod scales every 15 seconds.

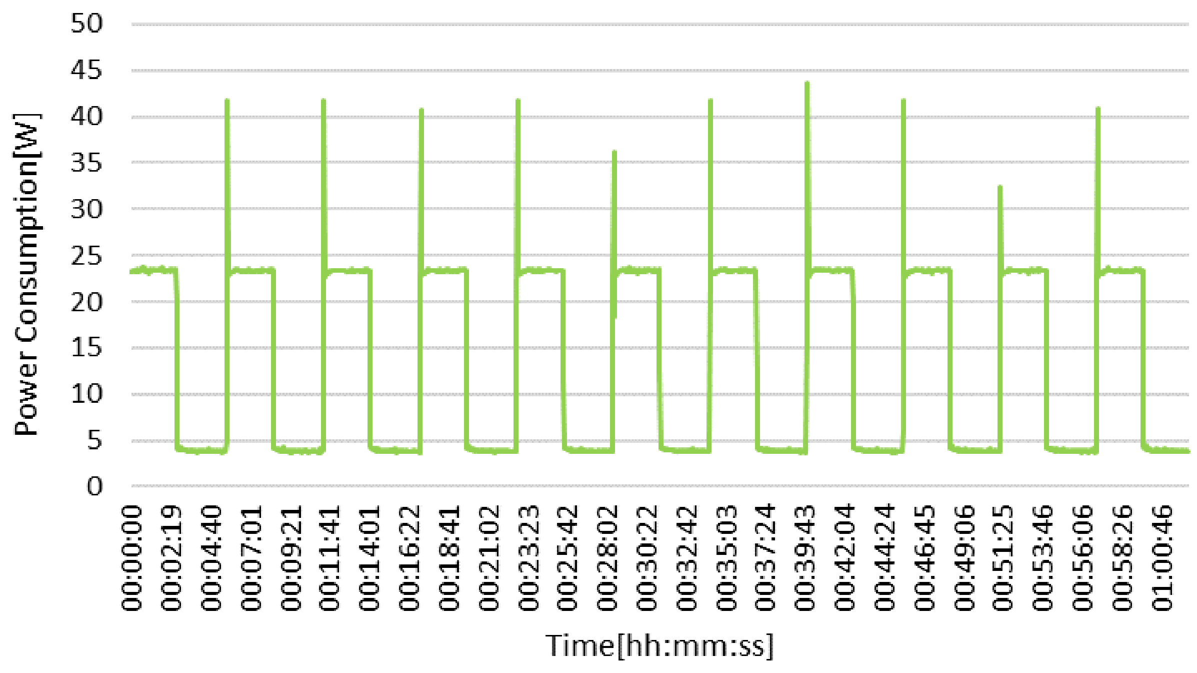


Figure 3-10: Power Consumption when the pod scales every 3 minutes.

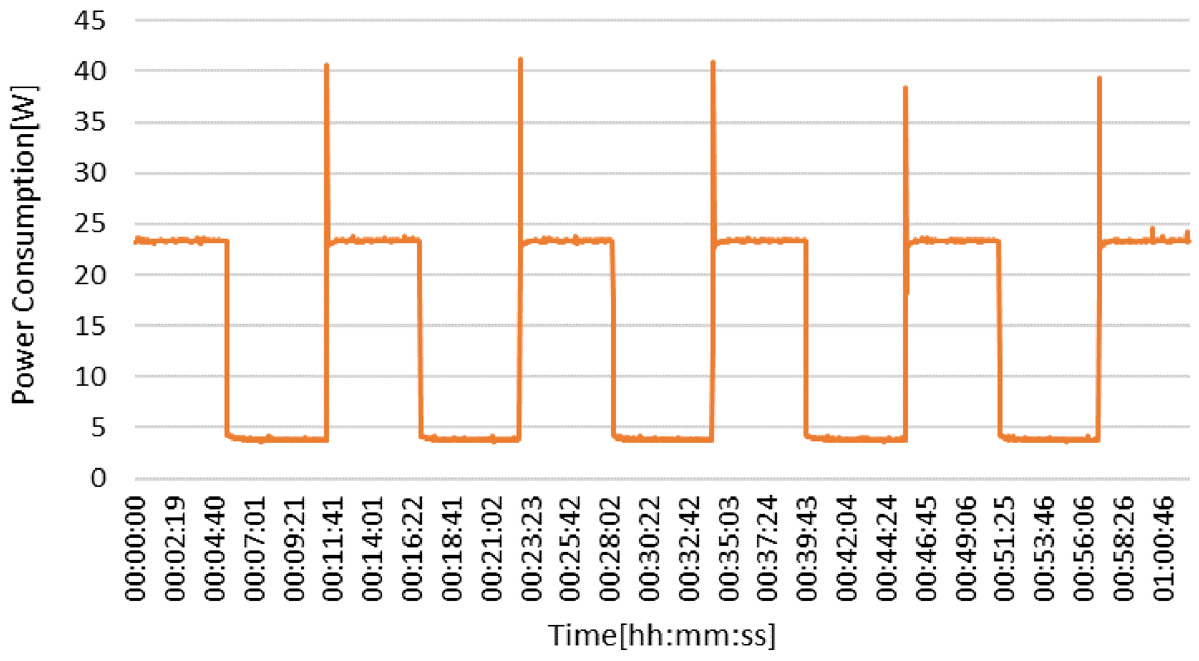


Figure 3-11: Power Consumption when the pod scales every 5 minutes.

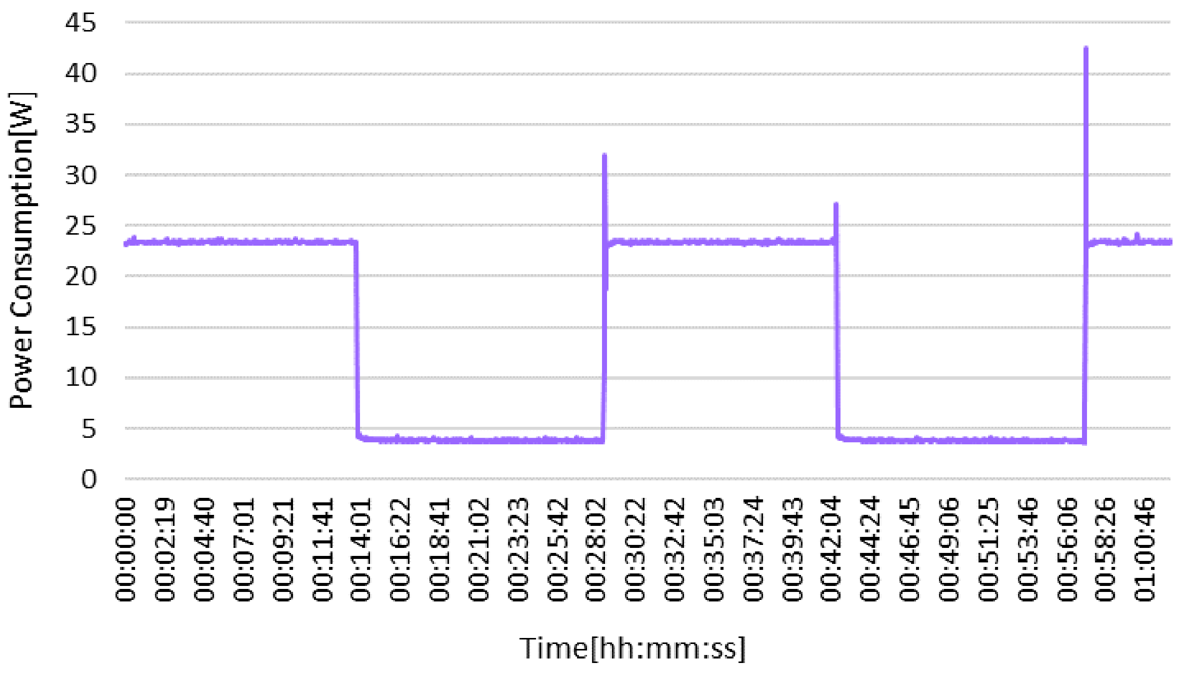


Figure 3-12: Power Consumption when the pod scales every 12 minutes.

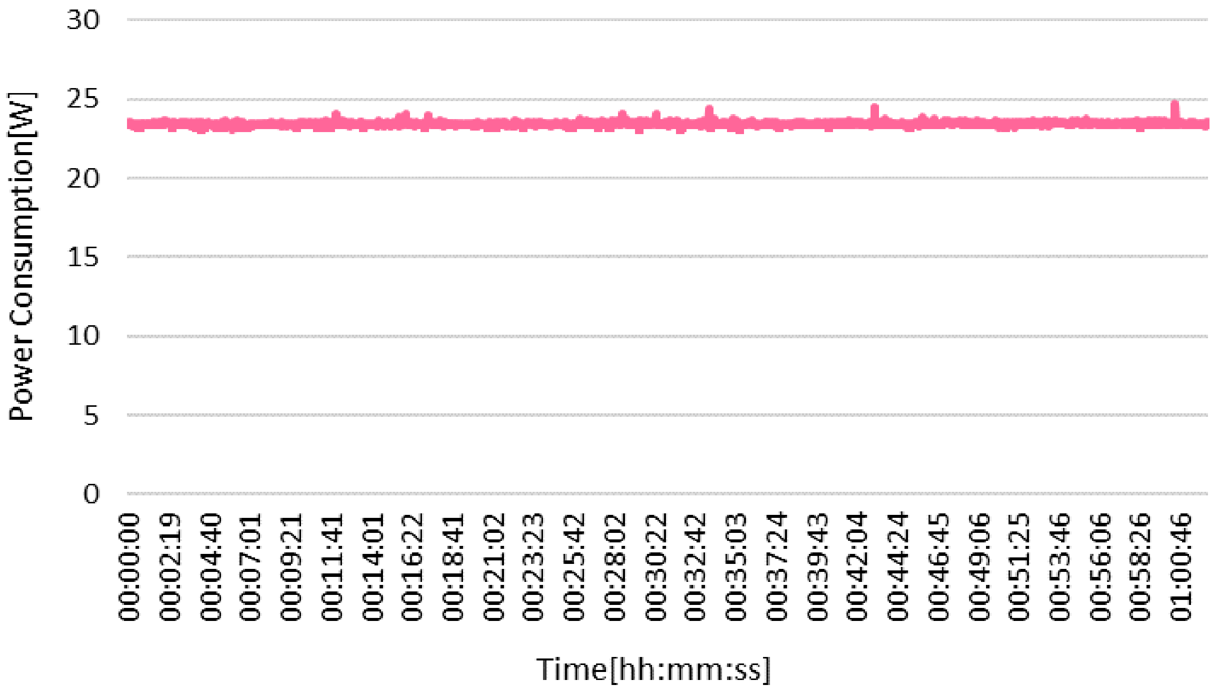


Figure 3-13: Power Consumption when the pod is active and receives traffic.

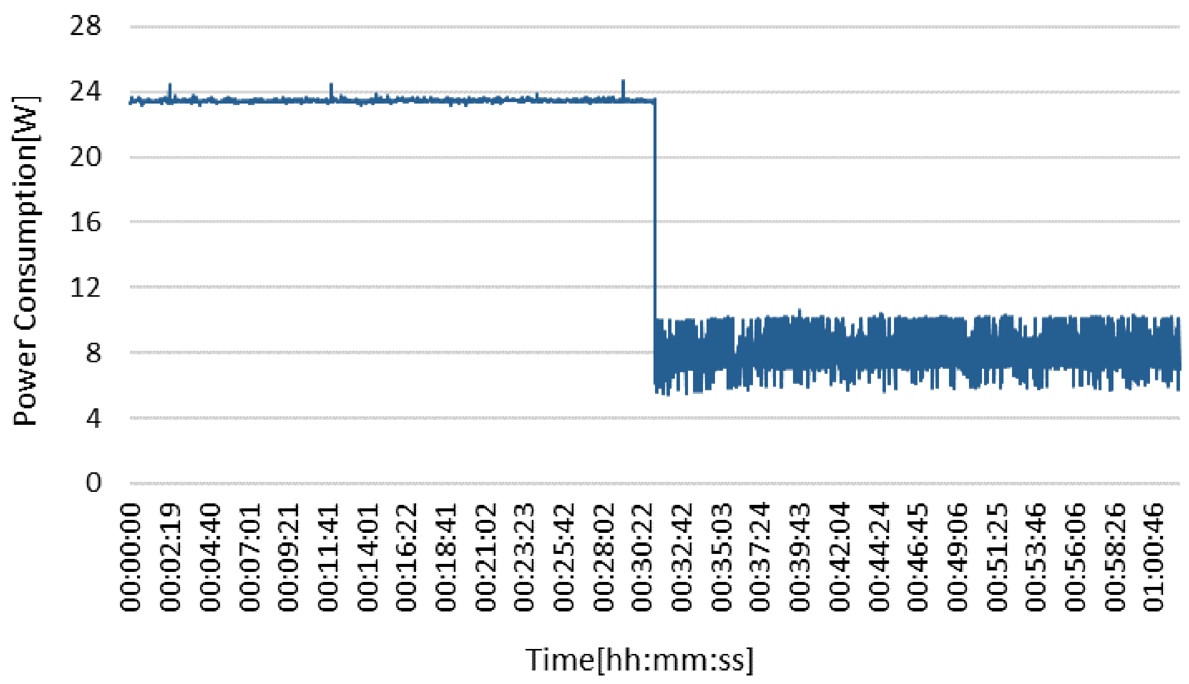


Figure 3-14: Power consumption when the pod is active, and traffic is stopped.

3.3 Infrastructure as a Code Mechanisms Based on the MetalCL

The MetalCL is devoted to the management and terraforming of the VIMs, operating systems and bare-metal resources available in a testbed. In the presence of this service, physical servers and hardware network equipment, as well as their operating systems, can be dynamically managed on demand. The conceptual framework behind the MetalCL is grounded in the paradigm of Infrastructure as a Code (IaC). In essence, the MetalCL serves as a versatile tool, enabling the composition of code to define, deploy, update, and destroy infrastructure elements essential for the realization of diverse projects. One of the prominent facets of the MetalCL's application spectrum is its utilization for the orchestration of 5Gbeyond-green initiatives. In this context, the MetalCL plays a pivotal role as an actuator, facilitating the dynamic alteration of states within the domain of bare-metal equipment. Furthermore, the MetalCL serves as a dedicated service for managing and terraforming bare-metal resources, encompassing physical servers and hardware network equipment to create IaaS and PaaS environments tailored to the specific requirements of 6G and 5G-beyond platforms. This capability includes overseeing operating systems on servers, configurations in network equipment, and installing complex distributed applications like Open-Stack and Kubernetes.

The MetalCL operates as an advanced infrastructure management system designed to optimize a wide range of hardware and software resources. Within its operational framework, the MetalCL is structured around several key components, each fulfilling distinct roles vital for cohesive resource management and allocation.

These pivotal components include delineated 'Zones,' which serve as segregated collections of hardware and software resources with specific functionalities. These zones can be characterized by unique levels of programmability, defining not only a diverse spectrum of resources but also the level of accessibility and programmability that can be performed on them: functionality within each zone spans from fundamental access to comprehensive management tasks, encompassing server selection, installation, and reconfiguration of instances.

Zones within MetalCL are tactfully implemented as encapsulated "plugins"; this architectural approach is pivotal in maintaining system modularity. By employing plugins, the MetalCL obviates the need for recurrent service recompilation and prevents the inadvertent introduction of unwarranted dependencies, thereby ensuring a coherent and adaptable system framework. The linkage between individual OpenStack or Kubernetes instances and projects is exclusive, establishing clear delineation in resource allocation and management.

The MetalCL interfaces with three external components: the MAAS server the Ansible engine and the NetCL.

The Metal-as-a-Service (MaaS) server, an open source project developed by Canonical[10], revolutionizes the management of individual bare-metal servers, bringing them in line with the administration of virtual machines within a cloud environment. This service enables the discovery, commissioning, deployment, and dynamic reconfiguration of extensive fleets of physical servers. With requirements as minimal as an IPMI-like system[11] and support for network boot operations through PXE standards [9], MaaS catalogues and manages these servers, offering functionalities akin to those in virtualized environments. Operating within MetalCL, MaaS utilizes its REST APIs to initiate bare-metal level changes, controlling power states and installing (on demand and as-a-Service) almost any operating system by properly configuring and administering the network(s) (mainly at the IP layer, while layer 2 interconnectivity and additional functions like gateways and firewalls are provided by the NetCL service). MaaS exposes a complete set of REST APIs, which are consumed by the MetalCL to trigger any changes at the bare-metal level.

Some key advantages of MaaS encompass automated remote operating system deployment, centralized monitoring, rapid provisioning, and tear down of bare-metal server configurations. It proves beneficial for environments necessitating frequent rearrangements of physical hardware, offering cloud-like agility to bare-metal setups. MaaS demonstrates its versatility across dynamic bare-metal infrastructure scenarios by treating physical servers as virtual resources. This approach infuses cloud-like flexibility into bare metal environments, efficiently handling deployment, modification, and reconfiguration of bare-metal setups. Integrating MaaS within MetalCL ensures adaptability and responsiveness to the evolving demands of infrastructure, making it invaluable for applications requiring frequent changes in server topology.

The Ansible Engine[12] plays a pivotal role within the MetalCL by driving any software installation, application and OS reconfigurations over the bare-metal servers installed by MaaS. This engine is the one that provides the MetalCL with the capability of installing software dependencies and installing and managing, in a zero-touch fashion, complex distributed software like OpenStack or Kubernetes over one or more servers, overseeing complex applications by assigning specific server roles, such as the number of controller or compute nodes. The zero-touch deployment model facilitated by Ansible ensures seamless and automated execution of tasks, significantly enhancing the efficiency and reliability of server-related operations within the MetalCL framework.

The NetCL serves as a key element for automated discovery, employing the LLDP protocol to uncover the physical topology. This capability allows it to access the command line or REST interfaces of networking devices, e.g., interconnection layer-2 managed switches (with VLAN or OpenFlow support), routers (optionally with the support of virtual routing functions) and firewalls, to enable the configuration of overlay networks. These networks not only facilitate the seamless hosting of complex platforms like OpenStack and Kubernetes but also actively manage interconnectivity among servers.

[10] https://maas.io/
[11] https://www.intel.it/content/www/it/it/products/docs/servers/ipmi/ipmi-second-gen-interface-spec-v2-rev1-1.html
[12] https://www.ansible.com/

Additionally, the NetCL assumes a pivotal role in network orchestration: it facilitates the automatic discovery of the physical topology and takes charge of interconnecting layer-2 switches, routers, and firewalls to establish overlay networks. This functionality is crucial in ensuring the efficiency of connectivity among servers, thereby significantly enhancing the robustness and stability of the entire network infrastructure.

3.3.1 Power Management Capabilities

At the heart of the MetalCL's resource management capabilities lie two fundamental pillars: the dynamic adjustment of CPU frequencies and the nuanced manipulation of C-States.

The importance of CPU frequencies lies in their direct effect on computational performance. Higher frequencies generally mean faster computations, making this parameter essential for a range of tasks. MetalCL's dynamic frequency adjustment allows users to tailor system performance based on their application's needs, promoting efficiency and responsiveness. MetalCL's dynamic CPU frequency management lets users balance the need for quick computations with the goal of minimizing energy use. This adaptability is valuable when computational requirements vary. Users can adjust the system's performance in real-time, responding to changes in workload intensity. MetalCL's flexibility in handling different workloads provides a practical solution for optimizing performance and energy efficiency in dynamic computing environments. In scenarios like cloud computing, where workloads can change unexpectedly, the ability to adjust CPU frequencies dynamically is crucial for resource allocation and cost-effectiveness.

C-States, denoting CPU power states, encapsulate a spectrum of power consumption and performance levels accessible to a CPU. MetalCL, empowers users with the unique capability to finely manipulate C-States. This granular control enables dynamic adjustments to individual CPU power states, responding adeptly to the ever-changing demands of diverse workloads.

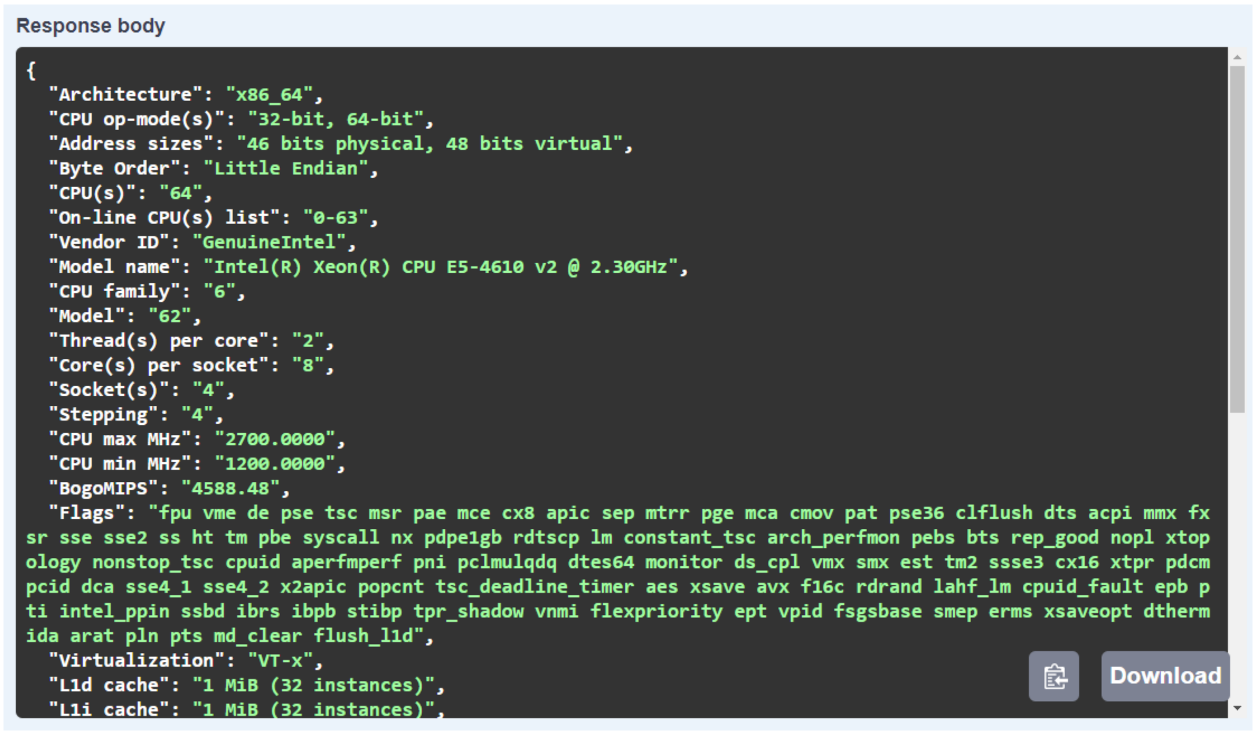
MetalCL's prowess extends beyond CPU frequency adjustment, as it seamlessly integrates the manipulation of C-States. C-States, ranging from C0 to Cn, represent a hierarchy where C0 signifies the highest-performance state, and ascending numbers (C1, C2, etc.) denote progressively deeper levels of power-saving states. This hierarchical structure allows CPUs to transition intelligently between states, aligning power consumption with the immediate processing requirements. The dynamic nature of C-State manipulation in MetalCL introduces a new dimension to power management, offering users a versatile tool to optimize energy efficiency and enhance hardware longevity.

The MetalCL has been recently extended with an API for retrieving information about the underlying hardware configuration. This architecture information encompasses critical details such as CPU model, number of cores and threads, cache sizes, CPU vulnerabilities, and more. Understanding these aspects of the CPU architecture aids in optimizing system performance, identifying potential vulnerabilities, and making informed decisions regarding hardware provisioning and management. In Figure 3-15, an illustrative example of the output is provided, while Figure 3-16 depicts a sample of representation of the available governors for a server.

The API facilitates the retrieval of the real time status of governors for each CPU core. This feature allows users to monitor and adjust the governor settings dynamically, ensuring efficient resource utilization. Figure 3-17 illustrates a sample of the status of governors for individual CPU cores. Additionally, the API offers access to the current frequency of each CPU core, enabling real-time monitoring of processor performance. This information allows users to analyze CPU usage patterns and make informed decisions regarding workload distribution and system optimization. An example showcasing the current frequency of CPU cores can be seen in Figure 3-18.

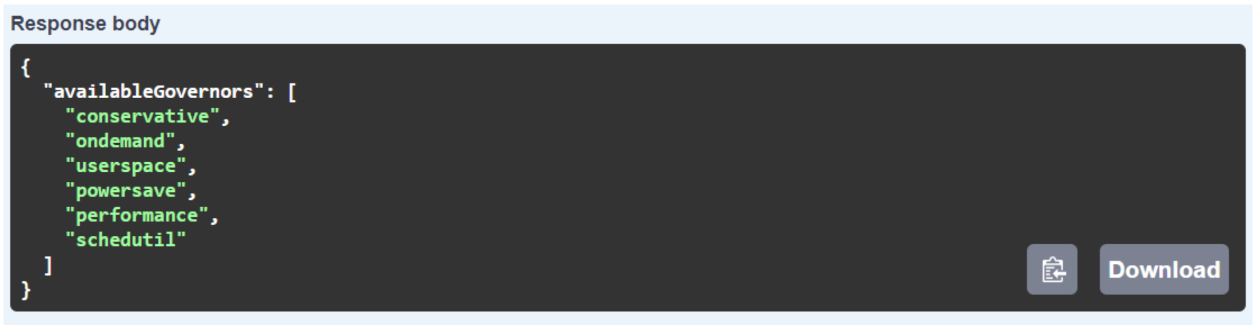
Two other relevant features are the monitoring of the available and current C-states for each CPU core, shown in Figure 3-19 and Figure 3-20, respectively. The former, along with detailed information about each state's characteristics and capabilities, facilitates fine-grained power management strategies., while the latter provides insights into power-saving behaviours and system efficiency.

Finally, users can utilize the API to obtain the percentage of time each CPU core has spent in a specific state, both for a specified interval or over the entire duration of system operation. This data allows for comprehensive analysis of CPU usage patterns and power consumption trends. Figure 3-21 presents an example of the percentage distribution of CPU core states.



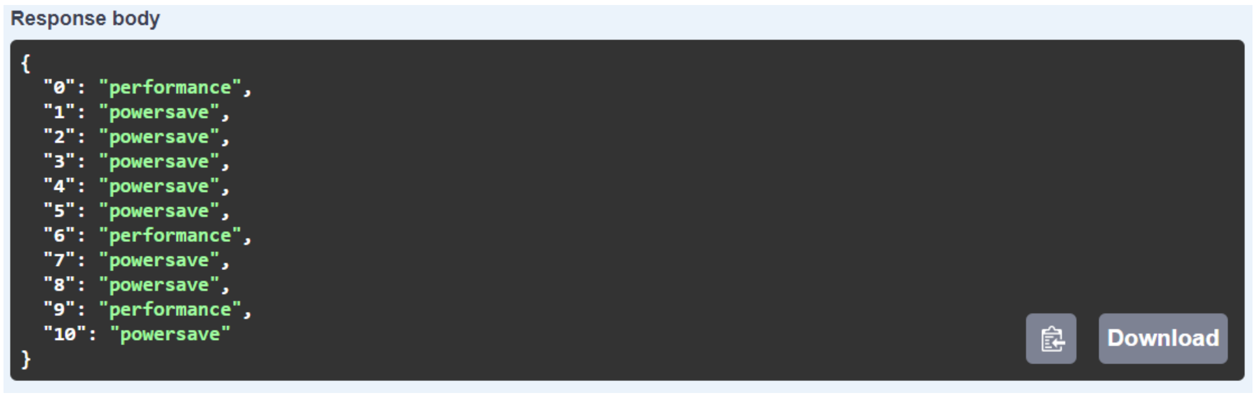
```
{
  "Architecture": "x86_64",
  "CPU op-mode(s)": "32-bit, 64-bit",
  "Address sizes": "46 bits physical, 48 bits virtual",
  "Byte Order": "Little Endian",
  "CPU(s)": "64",
  "On-line CPU(s) list": "0-63",
  "Vendor ID": "GenuineIntel",
  "Model name": "Intel(R) Xeon(R) CPU E5-4610 v2 @ 2.30GHz",
  "CPU family": "6",
  "Model": "62",
  "Thread(s) per core": "2",
  "Core(s) per socket": "8",
  "Socket(s)": "4",
  "Stepping": "4",
  "CPU max MHz": "2700.0000",
  "CPU min MHz": "1200.0000",
  "BogoMIPS": "4588.48",
  "Flags": "fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fx
sr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtop
ology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm
pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm cpuid_fault epb p
ti intel_ppin ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase smep erms xsaveopt dtherm
ida arat pln pts md_clear flush_l1d",
  "Virtualization": "VT-x",
  "L1d cache": "1 MiB (32 instances)",
  "L1i cache": "1 MiB (32 instances)",
```

Figure 3-15: Example output showcasing CPU architecture information retrieved through the API.



```
{
  "availableGovernors": [
    "conservative",
    "ondemand",
    "userspace",
    "powersave",
    "performance",
    "schedutil"
  ]
}
```

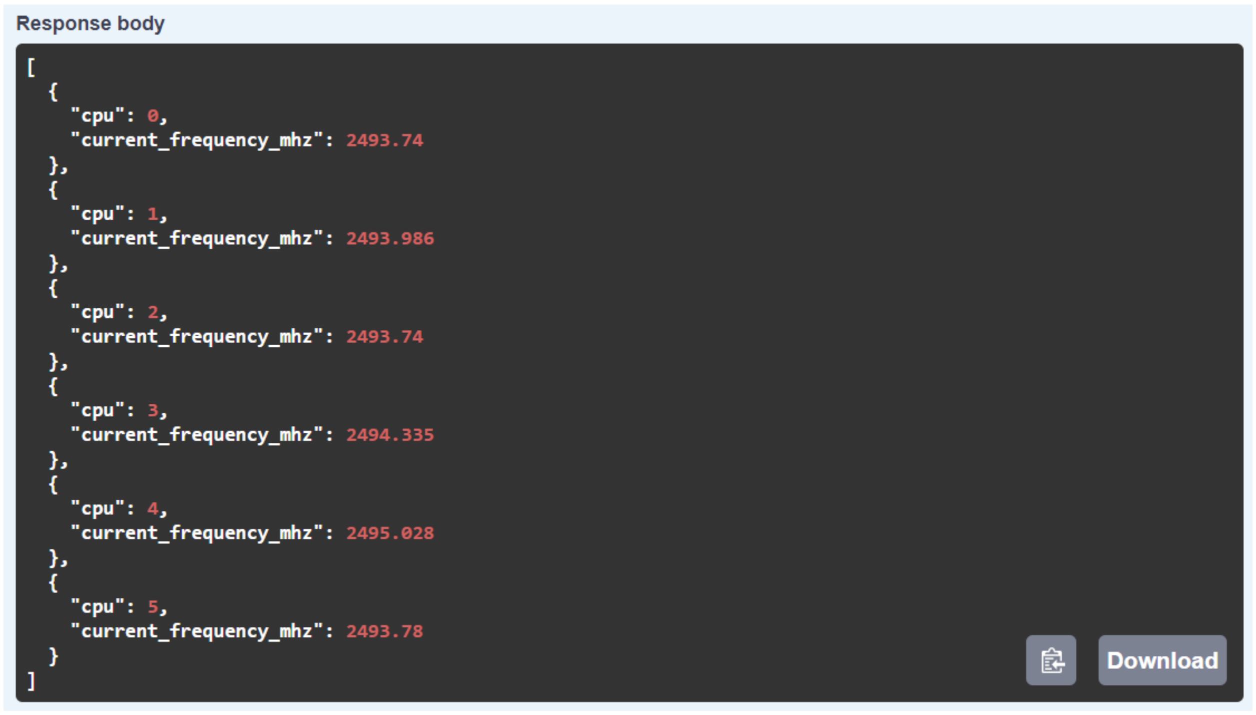
Figure 3-16: Representation of the available governors for CPUs retrieved using the API.



Response body

```
{
  "0": "performance",
  "1": "powersave",
  "2": "powersave",
  "3": "powersave",
  "4": "powersave",
  "5": "powersave",
  "6": "performance",
  "7": "powersave",
  "8": "powersave",
  "9": "performance",
  "10": "powersave"
}
```

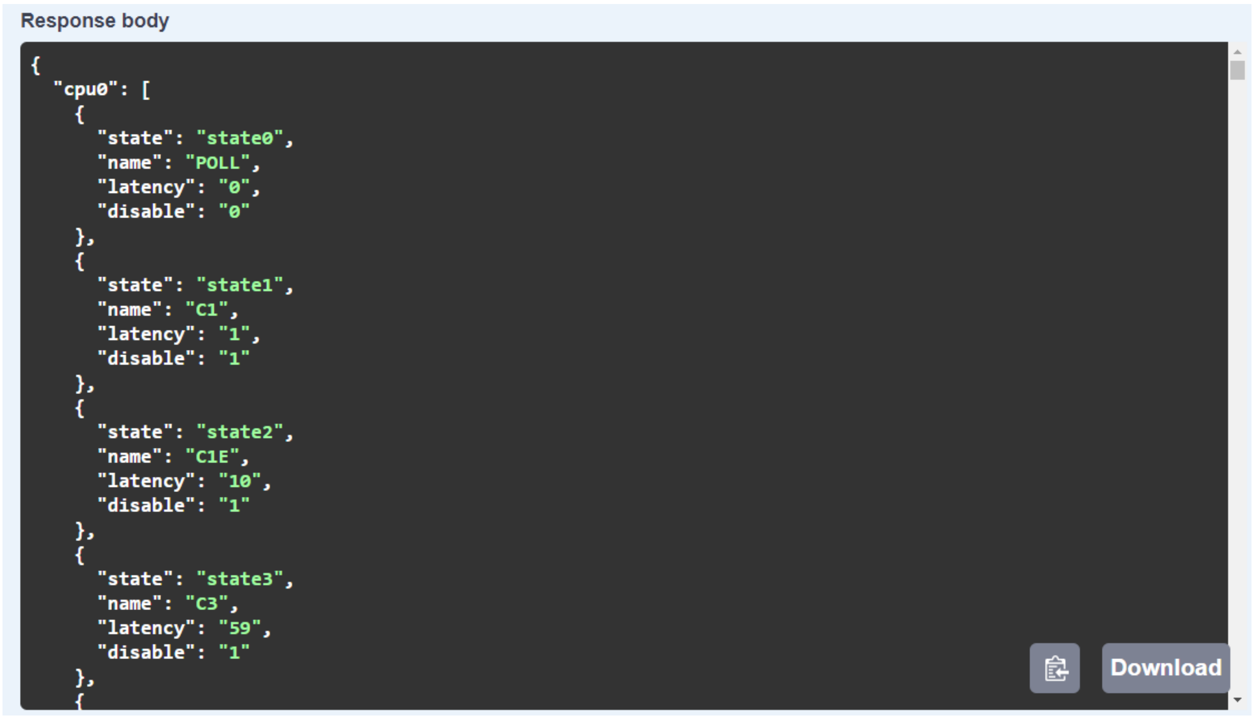
Figure 3-17: Monitoring of the status of governors for each CPU core through the API.



Response body

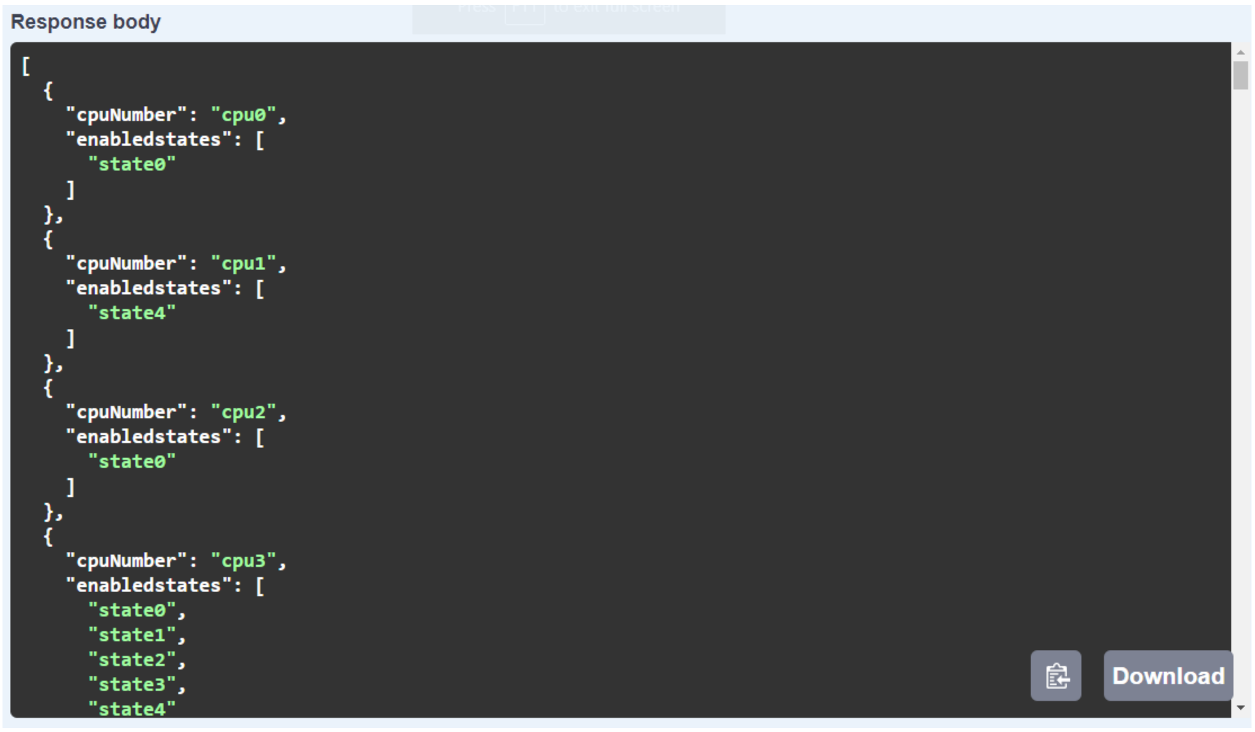
```
[
  {
    "cpu": 0,
    "current_frequency_mhz": 2493.74
  },
  {
    "cpu": 1,
    "current_frequency_mhz": 2493.986
  },
  {
    "cpu": 2,
    "current_frequency_mhz": 2493.74
  },
  {
    "cpu": 3,
    "current_frequency_mhz": 2494.335
  },
  {
    "cpu": 4,
    "current_frequency_mhz": 2495.028
  },
  {
    "cpu": 5,
    "current_frequency_mhz": 2493.78
  }
]
```

Figure 3-18: Illustration of the current frequency of each CPU core retrieved through the API.



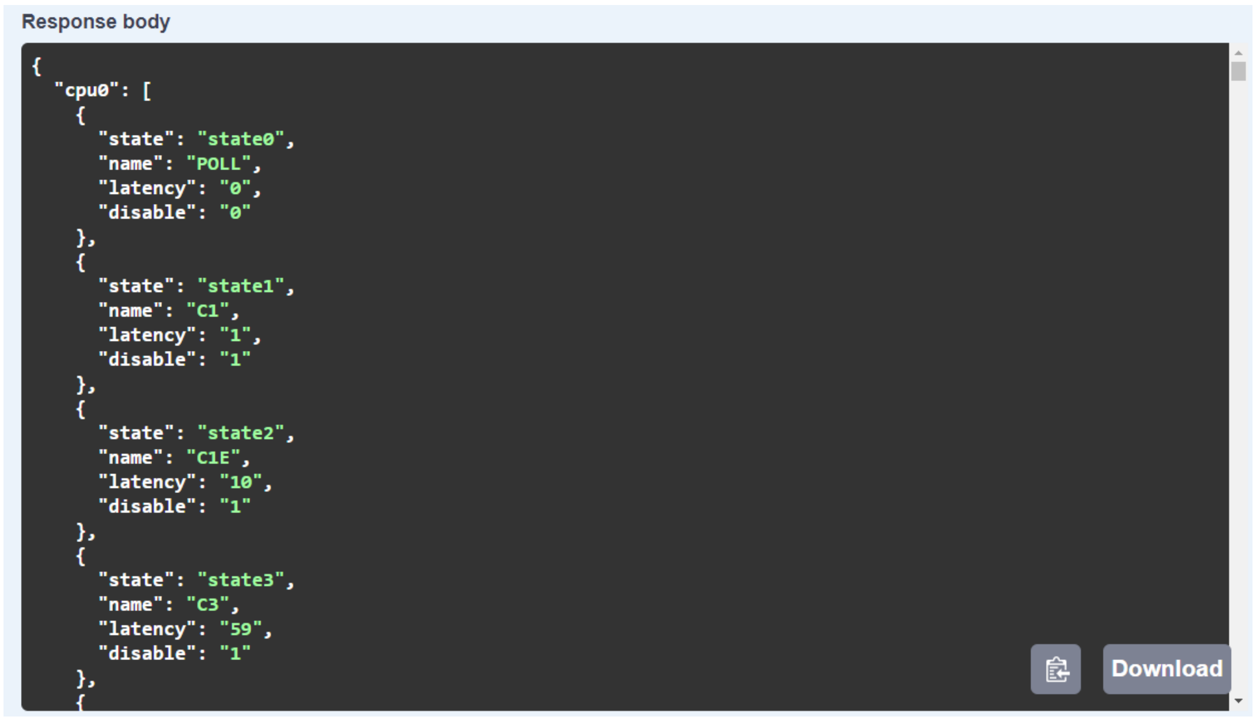
```
{
  "cpu0": [
    {
      "state": "state0",
      "name": "POLL",
      "latency": "0",
      "disable": "0"
    },
    {
      "state": "state1",
      "name": "C1",
      "latency": "1",
      "disable": "1"
    },
    {
      "state": "state2",
      "name": "C1E",
      "latency": "10",
      "disable": "1"
    },
    {
      "state": "state3",
      "name": "C3",
      "latency": "59",
      "disable": "1"
    },
    {
```

Figure 3-19: Visualization of the available C-states for each CPU core retrieved through the API.



```
[
  {
    "cpuNumber": "cpu0",
    "enabledstates": [
      "state0"
    ]
  },
  {
    "cpuNumber": "cpu1",
    "enabledstates": [
      "state4"
    ]
  },
  {
    "cpuNumber": "cpu2",
    "enabledstates": [
      "state0"
    ]
  },
  {
    "cpuNumber": "cpu3",
    "enabledstates": [
      "state0",
      "state1",
      "state2",
      "state3",
      "state4"
```

Figure 3-20: Real-time monitoring of the status of C-states for each CPU core through the API.

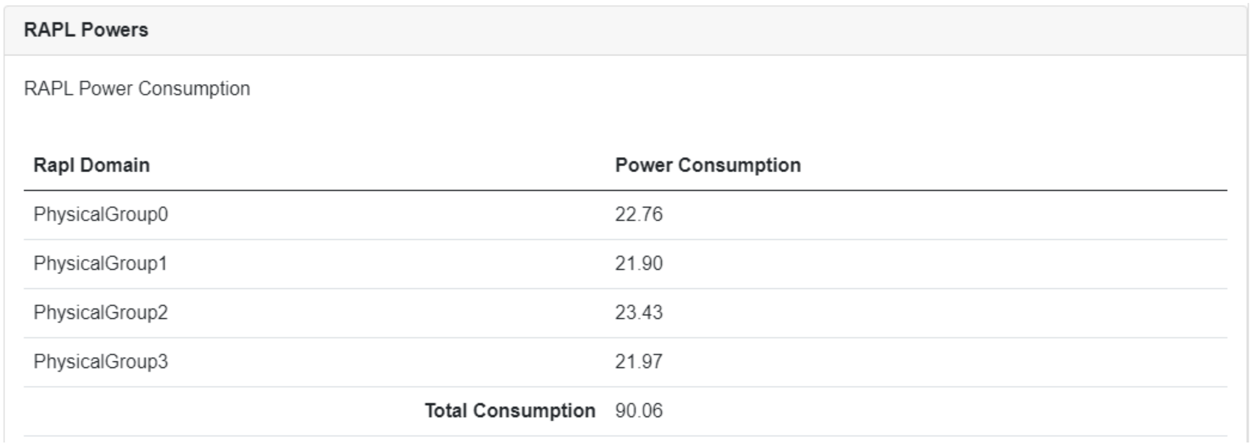


```
{
  "cpu0": [
    {
      "state": "state0",
      "name": "POLL",
      "latency": "0",
      "disable": "0"
    },
    {
      "state": "state1",
      "name": "C1",
      "latency": "1",
      "disable": "1"
    },
    {
      "state": "state2",
      "name": "C1E",
      "latency": "10",
      "disable": "1"
    },
    {
      "state": "state3",
      "name": "C3",
      "latency": "59",
      "disable": "1"
    },
    {
```

Figure 3-21: Example representation of the percentage distribution of CPU core states retrieved through the API.

The API also offers comprehensive capabilities for monitoring power consumption, leveraging three distinct methods to ensure accuracy and reliability in data acquisition. The RAPL (Running Average Power Limit) method provides insights into power consumption at the processor level, offering detailed information about energy usage patterns and fluctuations, as illustrated in Figure 3-22. Additionally, the API utilizes the IPMI (Intelligent Platform Management Interface) protocol to access power-related data from system hardware components, enhancing visibility into power consumption across various subsystems, as demonstrated in Figure 3-23. Furthermore, sensor-based measurements enable real-time monitoring of power usage at the hardware level, capturing fine-grained details about energy consumption in different system components, as depicted in Figure 3-24.

By employing multiple data collection methods, the API enhances the robustness and accuracy of power consumption monitoring, facilitating comprehensive analysis and optimization of energy usage. This multifaceted approach enables users to gain deeper insights into power consumption dynamics, identify inefficiencies, and implement targeted strategies to enhance energy efficiency and sustainability in computing environments.

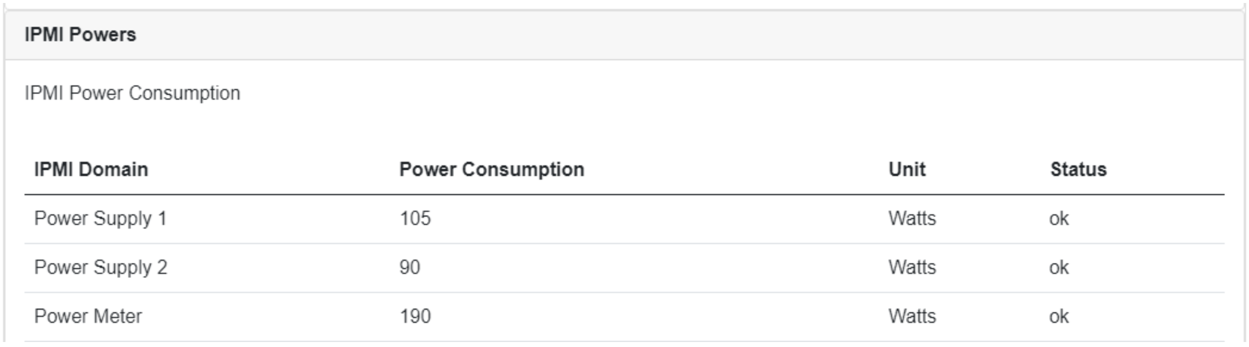


RAPL Powers

RAPL Power Consumption

Rapl Domain	Power Consumption
PhysicalGroup0	22.76
PhysicalGroup1	21.90
PhysicalGroup2	23.43
PhysicalGroup3	21.97
Total Consumption	90.06

Figure 3-22: Power consumption monitoring using the RAPL method.

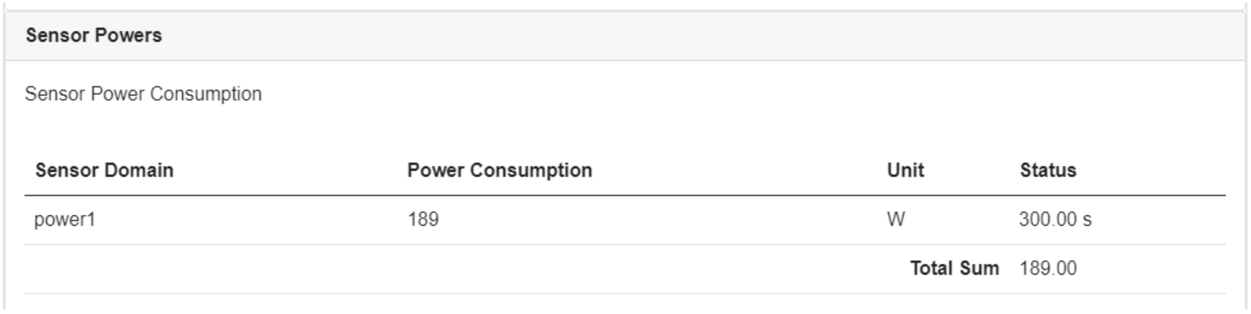


IPMI Powers

IPMI Power Consumption

IPMI Domain	Power Consumption	Unit	Status
Power Supply 1	105	Watts	ok
Power Supply 2	90	Watts	ok
Power Meter	190	Watts	ok

Figure 3-23: Power consumption monitoring utilizing the IPMI protocol.



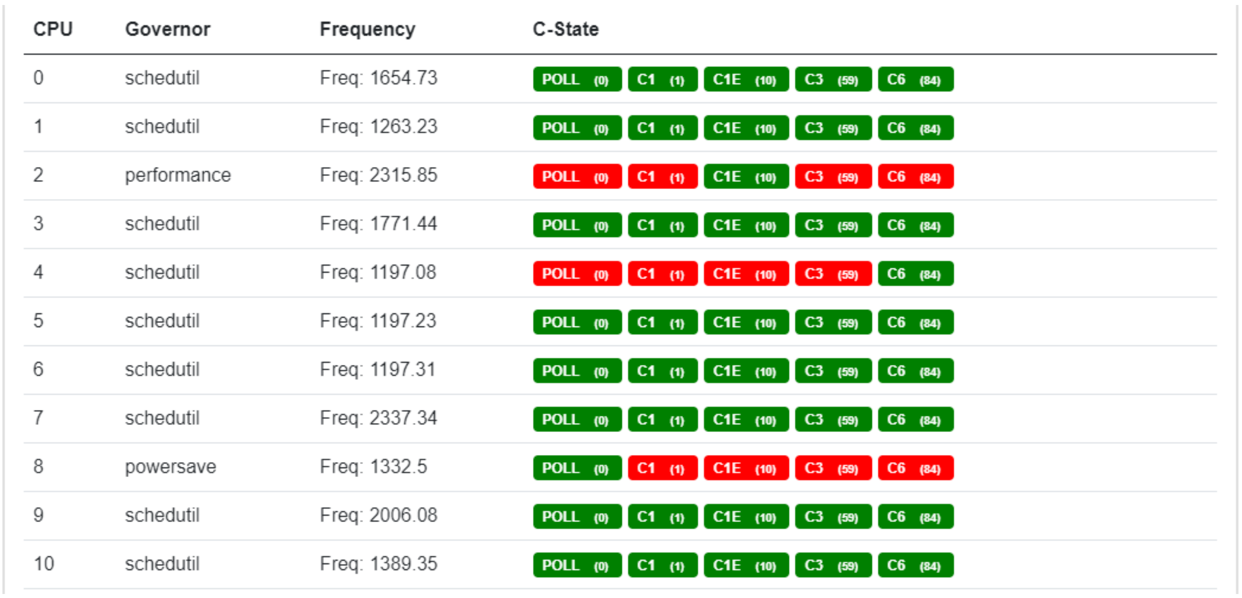
Sensor Powers

Sensor Power Consumption

Sensor Domain	Power Consumption	Unit	Status
power1	189	W	300.00 s
		Total Sum	189.00

Figure 3-24: Real-time monitoring of power usage through sensor-based measurements.

Enhancing user experience, a GUI is provided to offer a user-friendly interface for comprehensive system monitoring. Users can conveniently access a summary of the current governor assigned to each CPU core, providing insights into power management strategies and workload distribution. Additionally, real-time updates on the current frequency of CPU cores allow users to track performance fluctuations and optimize system resources effectively. Furthermore, the GUI provides visibility into the status of various available C-states, empowering users to fine-tune power-saving configurations for enhanced energy efficiency. Figure 3-25 offers a visual representation of these monitoring capabilities, showcasing the intuitive interface provided by the GUI.



CPU	Governor	Frequency	C-State
0	schedutil	Freq: 1654.73	🟩 POLL (0) 🟩 C1 (1) 🟩 C1E (10) 🟩 C3 (59) 🟩 C6 (84)
1	schedutil	Freq: 1263.23	🟩 POLL (0) 🟩 C1 (1) 🟩 C1E (10) 🟩 C3 (59) 🟩 C6 (84)
2	performance	Freq: 2315.85	🟥 POLL (0) 🟥 C1 (1) 🟩 C1E (10) 🟥 C3 (59) 🟥 C6 (84)
3	schedutil	Freq: 1771.44	🟩 POLL (0) 🟩 C1 (1) 🟩 C1E (10) 🟩 C3 (59) 🟩 C6 (84)
4	schedutil	Freq: 1197.08	🟥 POLL (0) 🟥 C1 (1) 🟥 C1E (10) 🟥 C3 (59) 🟩 C6 (84)
5	schedutil	Freq: 1197.23	🟩 POLL (0) 🟩 C1 (1) 🟩 C1E (10) 🟩 C3 (59) 🟩 C6 (84)
6	schedutil	Freq: 1197.31	🟩 POLL (0) 🟩 C1 (1) 🟩 C1E (10) 🟩 C3 (59) 🟩 C6 (84)
7	schedutil	Freq: 2337.34	🟩 POLL (0) 🟩 C1 (1) 🟩 C1E (10) 🟩 C3 (59) 🟩 C6 (84)
8	powersave	Freq: 1332.5	🟩 POLL (0) 🟥 C1 (1) 🟥 C1E (10) 🟥 C3 (59) 🟥 C6 (84)
9	schedutil	Freq: 2006.08	🟩 POLL (0) 🟩 C1 (1) 🟩 C1E (10) 🟩 C3 (59) 🟩 C6 (84)
10	schedutil	Freq: 1389.35	🟩 POLL (0) 🟩 C1 (1) 🟩 C1E (10) 🟩 C3 (59) 🟩 C6 (84)

Figure 3-25: Visual representation of the provided GUI.

3.3.2 Results

The aim of this testing is to analyze the power consumption of a server under different configurations of c-states and governors. Specifically, we want to understand how altering these settings impacts power usage both under normal CPU loads and under maximum CPU stress.

In the initial configuration, the C-state is set to State0, and the governor is set to Performance. The system is operating under normal load conditions. Upon changing the governor to Powersave, there is a noticeable decrease in power consumption from 371.24 W to 249.18 W, as depicted in Figure 3-26. This demonstrates the impact of governor settings on power usage when the system is not fully loaded.

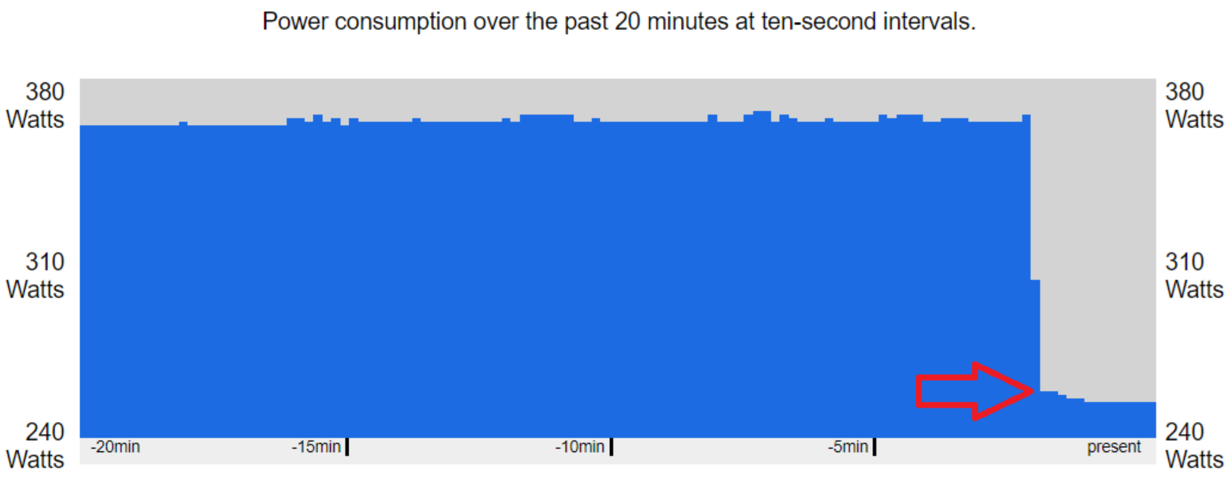


Figure 3-26: Comparison of power consumption before and after changing the governor to Powersave with normal system load.

With the governor set to Powersave and increasing the system load to 100%, the power consumption rises to 263.45 W, as shown in Figure 3-27. This illustrates that even with a Powersave governor, power consumption increases under full load conditions.

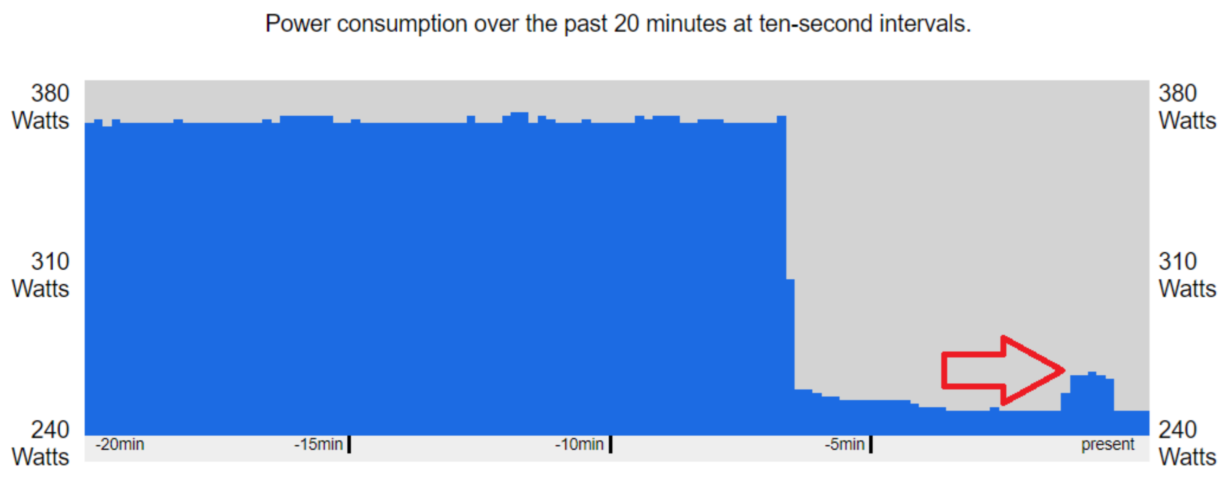


Figure 3-27: Power consumption with the Powersave governor under full system load, illustrating increased power usage compared to the previous configuration.

After changing the governor back to Performance while maintaining the 100% system load, the power consumption further increases to 407.23 W, as seen in Figure 3-28. This emphasizes the role of the governor in influencing power consumption under varying workloads.

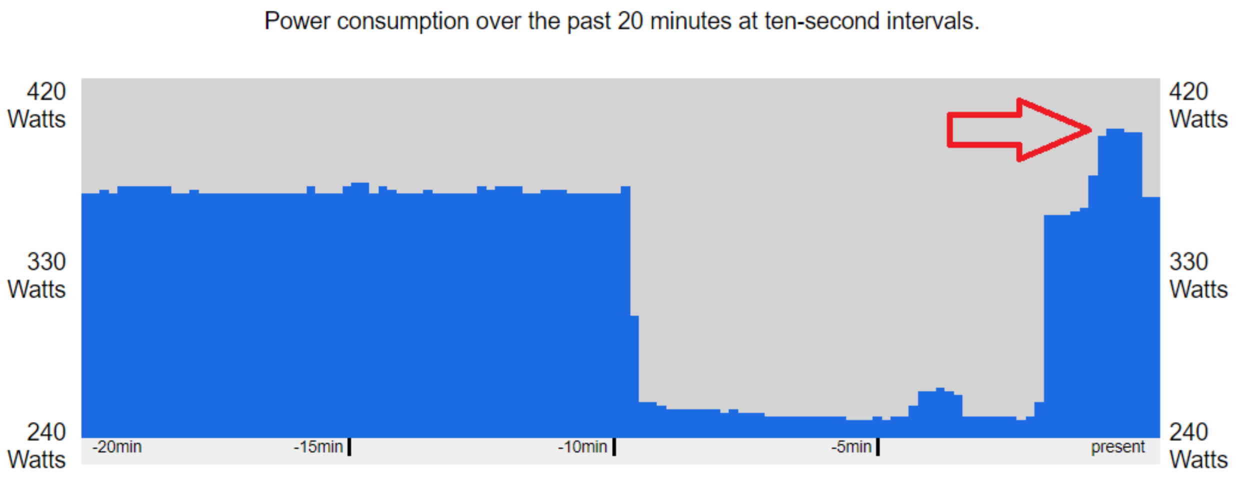


Figure 3-28: Power consumption spikes after reverting the governor back to Performance while maintaining full system load.

When the C-state is changed to All while the system is not under full load, the power consumption decreases significantly to 177.43 W, as depicted in Figure 3-29. Enabling all C-states allows for better power management when the system is idle.

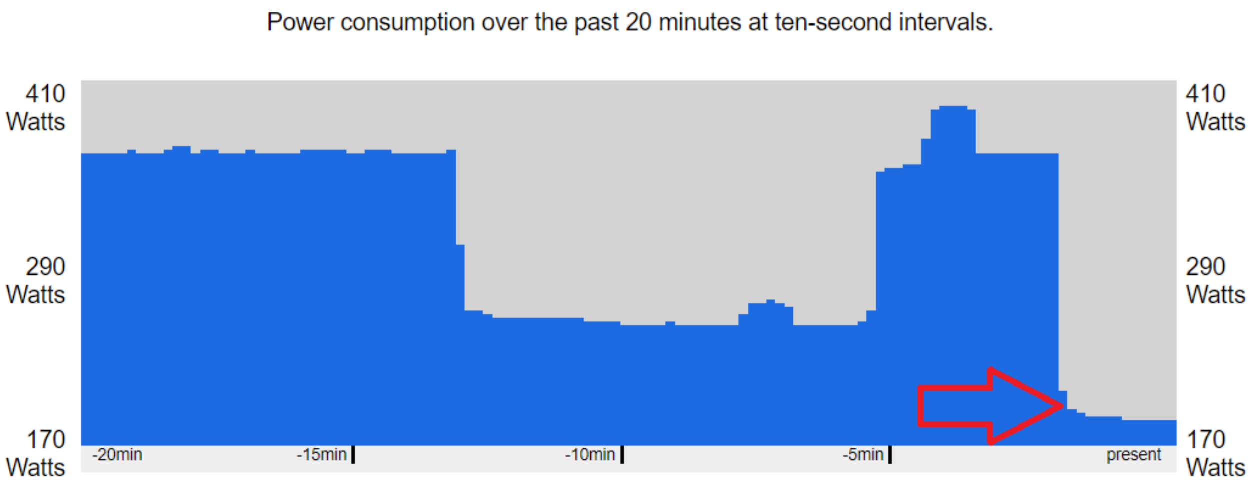


Figure 3-29: Significant decrease in power consumption with all C-states enabled under normal system load conditions.

Even with all C-states enabled, when the system load is increased to 100%, the power consumption rises notably to 407.11 W, as shown in Figure 3-30. This highlights that while C-states can help in reducing power consumption during idle states, they may not have a significant impact when the system is under heavy load.

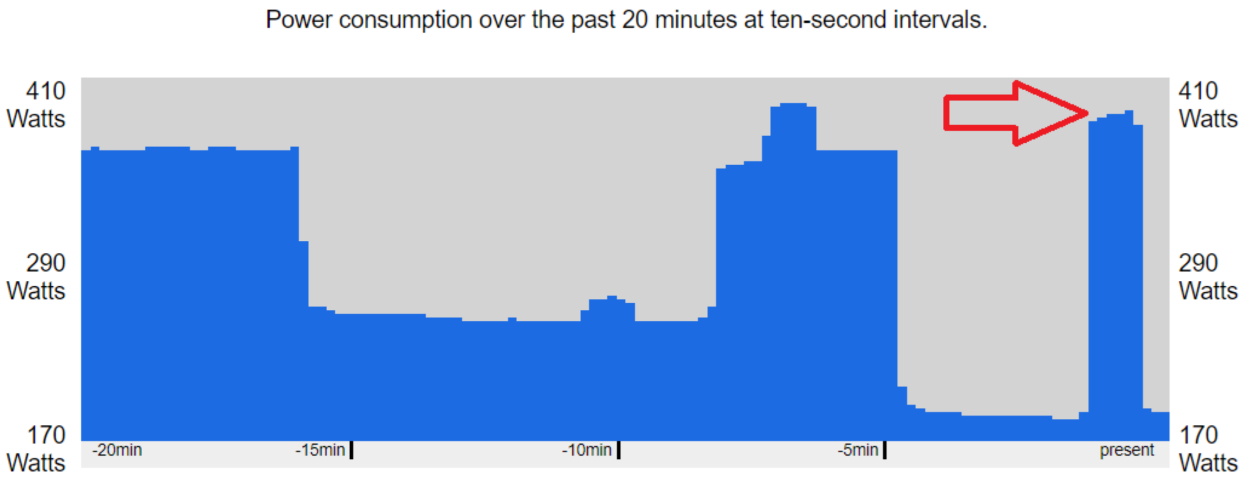


Figure 3-30: Power consumption increases notably under full system load even with all C-states enabled.

With the governor set to Powersave and the system not fully loaded with all C-states enabled, the power consumption is 172.13 W, as indicated in Figure 3-31. This likely demonstrates a lower power usage compared to the Performance governor under similar conditions.

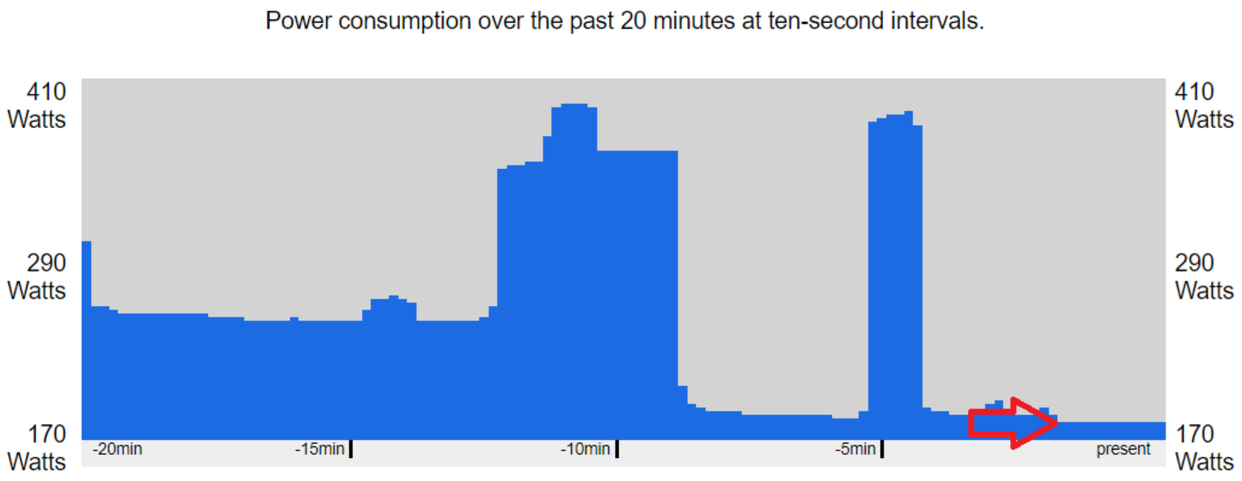


Figure 3-31: Lower power usage observed with the Powersave governor and all C-states enabled under normal load conditions.

Under 100% system load with the governor set to Powersave and all C-states enabled, the power consumption is 261.25 W, as shown in Figure 3-32. This figure illustrates the power consumption increase compared to the previous figure due to the higher workload. But as the governor is set to Powersave, we see that this increase is not as much as the time that the governor is set to Performance.

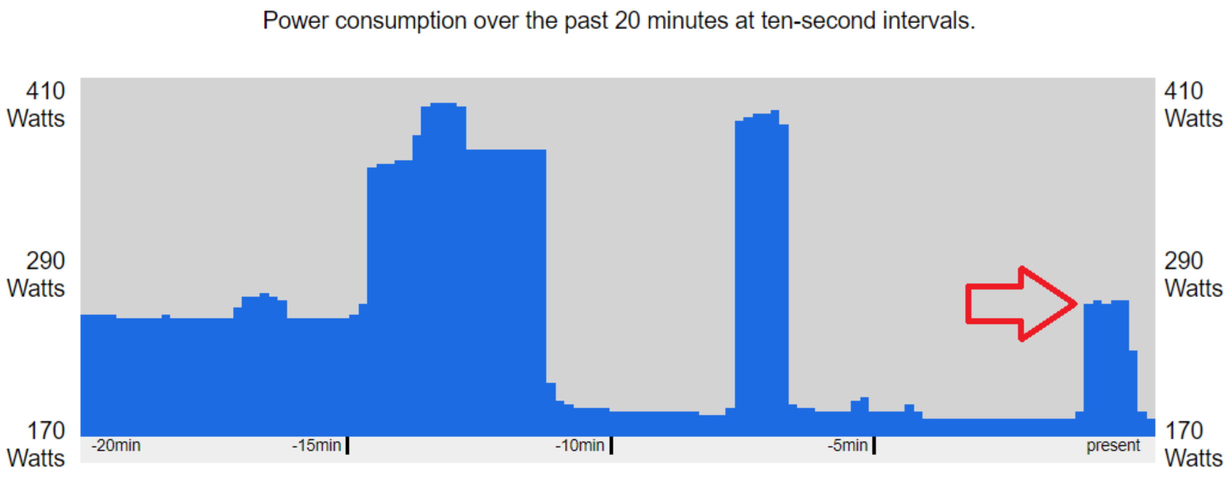


Figure 3-32: Increased power consumption under full system load with the Powersave governor and all C-states enabled compared to the previous configuration.

When the C-state is changed to only State2 while the governor remains as Powersave, the power consumption reaches to 193.72 W, as observed in Figure 3-33. This highlights the influence of specific C-state configurations on power efficiency.

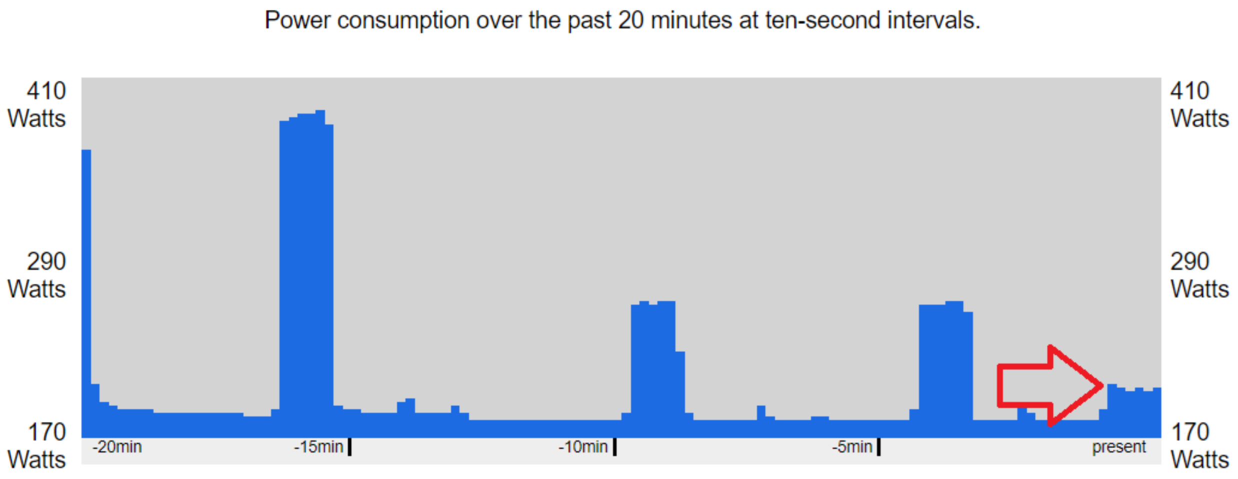


Figure 3-33: Power consumption with specific C-state configuration (only State2) and Powersave governor under normal load conditions.

By setting the governor to Ondemand while C-states are set to all and with the system load at 100%, the power consumption increases to 406.03 W, as shown in Figure 3-34. This demonstrates that the Ondemand governor can reach maximum power consumption levels when the workload is at 100%, similar to the Performance governor configuration.

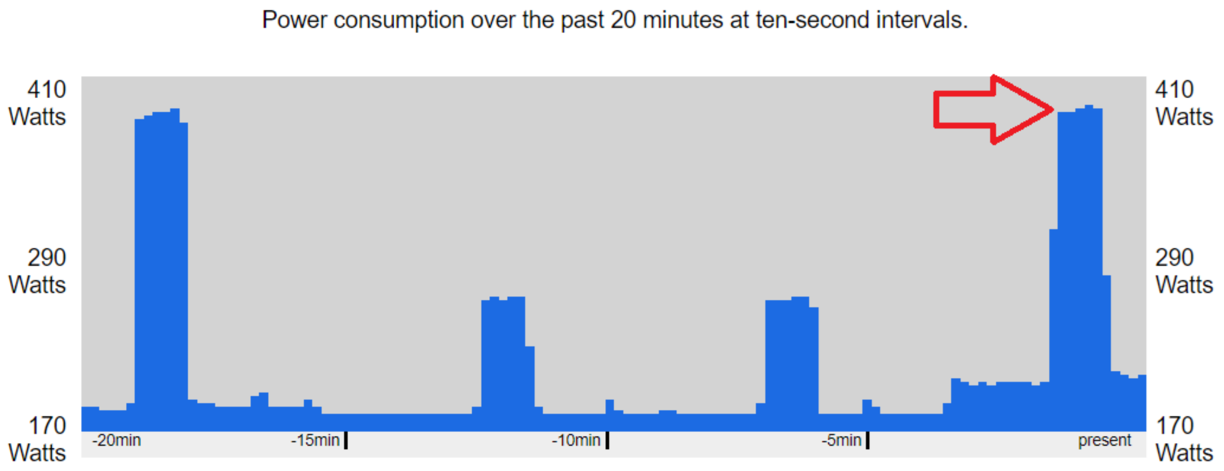


Figure 3-34: Power Consumption with Ondemand Governor and All C-states Enabled Under Full System Load.

3.4 ZeroOps and Continuous Automation Based on the NFV Convergence Layer (NFVCL)

The NFVCL is a network-oriented meta-orchestrator, specifically designed for zeroOps and continuous automation. It can create, deploy and manage the lifecycle of different network ecosystems by consistently coordinating multiple artefacts at any programmability levels (from physical devices to cloud-native microservices).

In detail, a network ecosystem like the one in Figure 3-35 is meant to be a complete functional network environment, such as a 5G system, an overlay system for network cybersecurity or a simple application service mesh. For their nature, these environments are realized through heterogeneous Network Functions (xNFs – i.e., Physical NF, Virtual NF and cloud-native Kubernetes NF), which are usually to be realized over highly distributed infrastructures. More specifically, every network ecosystem can be thought of as a graph $\mathcal{G} = \langle \mathcal{V} \cup \mathcal{N}, \mathcal{E} \rangle$, where the vertexes are composed of the sets of xNFs instances $\mathcal{V}$ and the sets of interconnection networks $\mathcal{N}$, while the $\mathcal{E}$ represents the interconnectivity edges between xNFs and networks.

As defined in the ETSI NFV standard, xNFs are managed by the NFVO through end-to-end Network Service Instances. Every Network Service can include one or more xNF instances, and it is meant to be deployed over a single geographical facility, which may correspond to a computing facility and/or a physical device (e.g., a gNodeB, an O-RAN Radio Unit, a P4 switch, etc.).

The graph $\mathcal{G}$ of a network ecosystem, represented in Figure 3-36, is meant to be annotated with "anchors" that represent the placing/binding of the ecosystem endpoints over the physical infrastructure topology. An anchor can be associated to a network in the $\mathcal{N}$ set, or to a PNF to be instantiated over a physical device.

Moreover, Figure 3-35 and Figure 3-36 highlight the support for having different levels of virtualization which can be exploited by the NSs. Platform as a Service (PaaS) allows the deployment of KNFs, Infrastructure as a Service (IaaS) of VNFs and finally Metal as a Service (MaaS) allows to bypass virtualization and to deploy services directly on the Hardware (e.g., a Kubernetes bare-metal cluster).

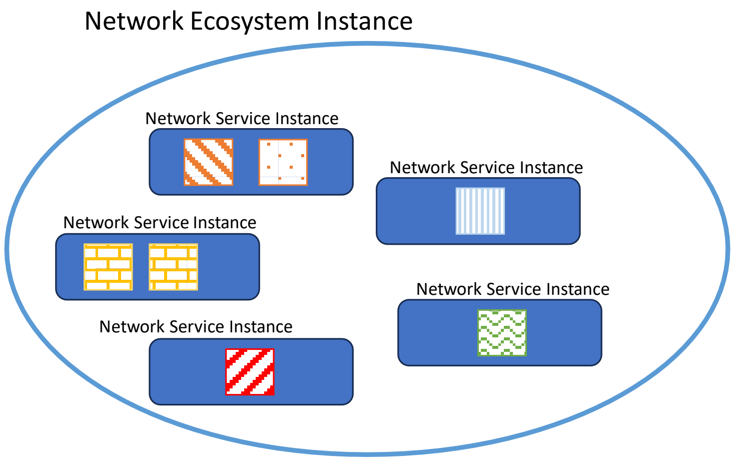


Figure 3-35: A Network ecosystem instance composed of 5 Network Services made up of a variable number of xNFs.

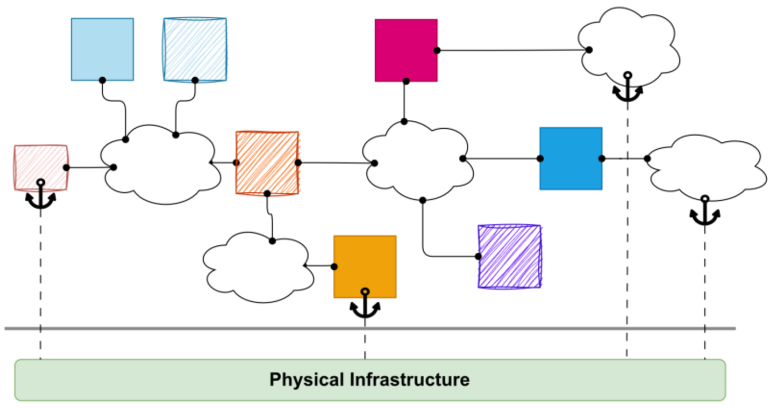


Figure 3-36: The graph of a network ecosystem with anchor points highlighting the link between the xNFs and the physical infrastructure.

3.4.1 The Network Ecosystem

The NFVCL has been built over a modular and flexible architecture that can be easily extended to support new xNF and ecosystems. At the foundations of this architecture, the metamodel in Figure 3-37 has been specifically designed to augment extensibility and flexibility and to drive clear interaction patterns among the different internal modules during LCM operations.

Still with reference to Figure 3-37, every ecosystem instance is built through a Blueprint, which on its turn falls into a Category. The Blueprint Category corresponds to the high-level network ecosystem function type, like a 5G system, a network security tool-chain, etc. The Blueprint is meant to support ad-hoc operations for specific implementations falling into that Category. For instance, the NFVCL currently provides 4 different 5G system implementations, based on different open-source projects, namely Free5GC, Open5GS, OpenAir Interface and SD-Core.

The **Blueprint Category** allows to have a homogeneous north-bound interface against the different implementations available for an ecosystem, since it defines a single input meta-data model (including the possible ecosystem end-points) and the associated ecosystem-level LCM methods. For example, the 5G System Blueprint Category exposes operations to add/remove/reconfigure RAN over specific geographical areas, to create/modify/destroy network slices, etc., and it fixes the end-points to be physical devices like base stations or O-RAN radio units, and networks to be used as 5G DNN.

A **Blueprint** provides the implementation-specific means to support the Category methods and to translate the metadata model into sets of NSIs and xNFs, interconnected and running with coherent (but implementation-specific) configurations. To this end, Blueprints defines the template of the ecosystem internal topology, as well as the specification of the internal procedures to be executed for every supported Category method. These internal procedures are realized as saga pattern interactions among specific NFVCL modules.

A Blueprint contains a lot of data that can be categorized in:

- Status: contains information on the status of resources (like the list of interfaces with the relative IPs).
- Configurators: the list of configurators (status included) that are created and used by the Blueprint (Day-0, Day-2, Day-N).
- Topology: the information on the topology in which the Blueprint is deployed.

The code of a Blueprint class is the one managing how, and in which order, Resources are generated. The Blueprint instance is also managing Day-2 operations like adding, updating and deleting a node from the blueprint instance. The new Blueprint system abstracts the concept of Provider, offering a uniform set of functions to every type of Blueprint. These functions are offering the tools for the LCM of resources composing the specific instance of that type of blueprint. Since a Blueprint can be composed of both VMs and K8s resources, the provider interaction is not limited to one, but we can interact with several providers.

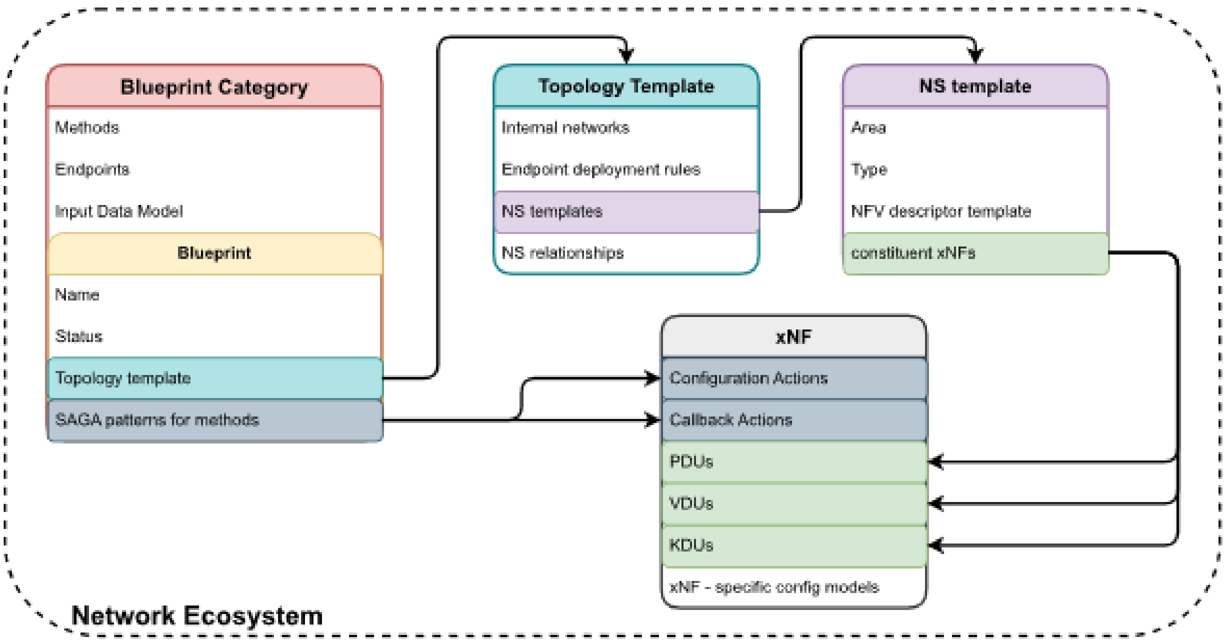


Figure 3-37: Network ecosystem metamodel.

The **topology template** defines the $\mathcal{G}$ graph pattern including the templates of internal network and of NSIs that can be applied, and their possible relationship bindings. Everything that is contained in the Topology is used by the NFVCL to manage the lifecycle of Blueprints. For example, when a Blueprint is deploying VMs, the VIM to be used is identified using the VIM data saved in the Topology.

Finally, the **xNFs metadata** model plays a key role in the NFVCL architecture. It defines not only the specific physical/virtual/Kubernetes deployment units to be used to materialize NS templates, but it also defines the implementation-specific methods and callbacks that can be executed on an xNF, and the models of its configuration. In other words, xNF templates represent a sort of glue between NFV-driven LCM operations to instantiate or remove artefacts from the ecosystem (e.g., creating a RAN NS in a new area), and management operations affecting the configuration of running xNFs (e.g., add a new 5G subscriber, add a new policy, etc.).

Each of these operations might include a variable number of different actions to add NSI instances (Day 0 and 1 actions), to change the configuration settings of xNFs and to retrieve information from the deployed xNFs (Day 2 actions), as well as to remove one or more deployed NSIs.

3.4.2 The NFVCL Architecture

The NFVCL internal architecture (Figure 3-38) encompasses the meta-models introduced in the previous Section. A first module, named **NFVCL North Bound Interface** aims at exposing CRUD REST APIs for ecosystem LCM trough the methods defined in the Blueprint Category meta-models that are available and onboarded to the NFVCL. Among these methods, the ecosystem creation and deletion are mandatory (and correspond to HTTP POST and DELETE messages).

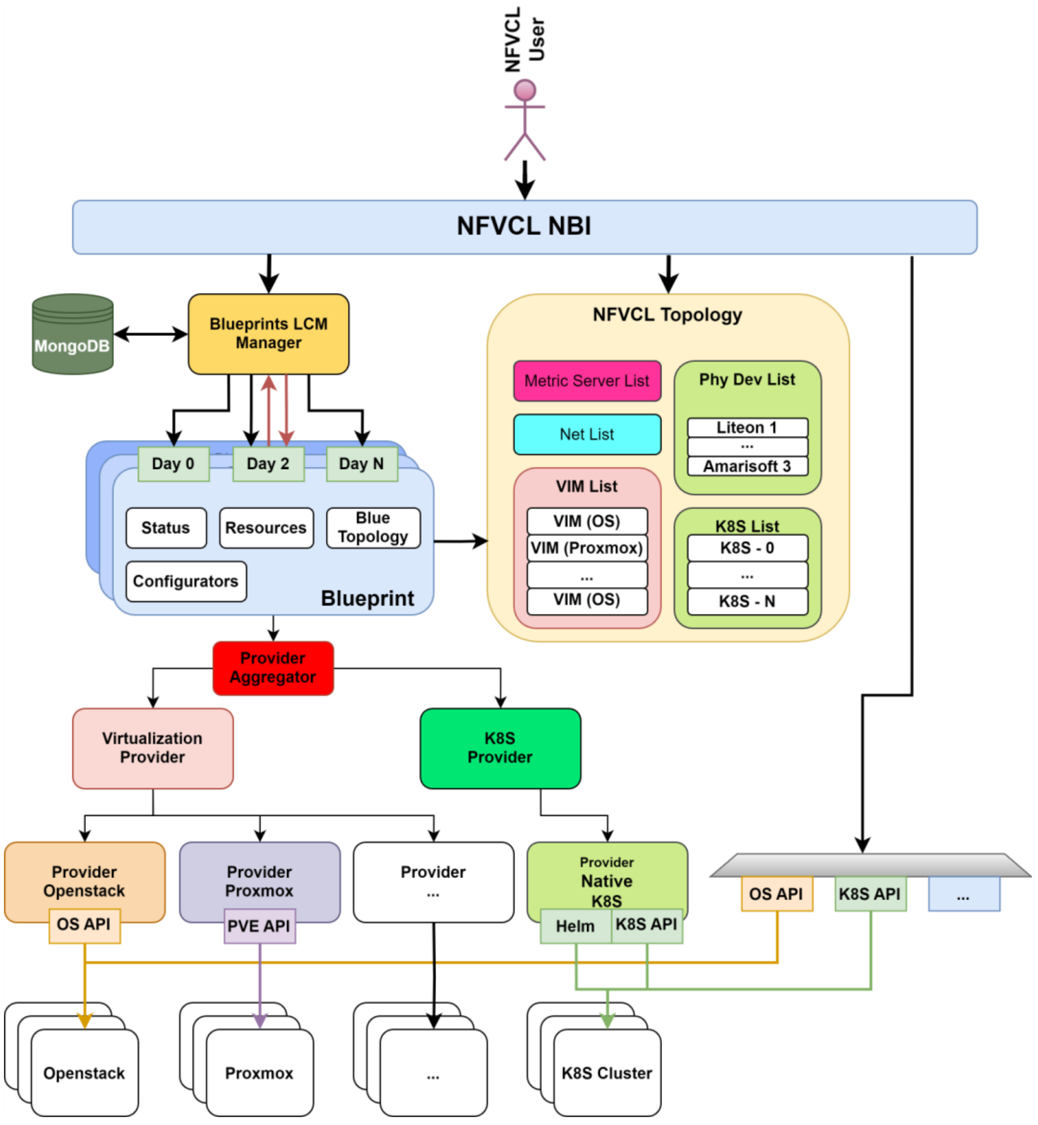


Figure 3-38: The NFVCL internal architecture.

The **NFVCL Topology** offers resources needed by the ecosystems through several lists. The *VIM List* offers the possibility to deploy VMs to the NFVCL. This functionality is used by Blueprints to deploy what is requested by the user (e.g., a K8s cluster with 1 Controller and 2 Workers for a total of 3 VMs). We can have a list of VIMs associated to an area, in this way, the user can select the area in which the Blueprint will be deployed. In the case of multiple VIM for the same area, the first one is used. The *K8s List* contains all the K8s clusters that can be used by Blueprints to deploy Helm Charts. As for the VIM List, every K8s cluster is associated to an area and the user can select the one to use. The *Net List* is used to keep track of the networks available in the VIMs. Networks can be added manually, if already present, and can be also added and created by a Blueprint, if needed. The *Metric Server List* contains Prometheus instances that can be used to configure metrics exporters on Blueprint Resources. Finally, the *Physical Device List* contains physical network functions, for example HW-based UPFs.

The **Blueprint Lifecycle Manager** takes care of all the requests towards blueprints, from creation to Day-N operations. This component also allows for cross-blueprint interaction (even creation and deletion).

The **MongoDB** database stores all dynamic vital information for the operation of the NFVCL. The two main collections to be saved are the status for the topology and, for every instantiated blueprint, the status and the topology template. The saved data of the blueprint can also include past actions, like executed LCM primitives.

The **Blueprint** handles and serializes incoming LCM initialization/change requests on the ecosystem. In particular, it is in charge of binding any supported blueprint category method into a coordinated set of multiple implementation-specific operation requests against resources in the topology, the LCM of NFV Network Service Instances (NSI), or configuration changes within one or multiple xNFs.

The **Provider Aggregator** is a layer in charge of abstracting the virtualization and K8s providers in a single interface accessible from the blueprint to create and configure its resources. In more details, the Virtualization/VM Provider is used to deploy the same blueprint type on different VIM types (currently OpenStack and Proxmox) without the need to adjust the blueprint code, and the K8s/Helm Provider embodying the same role for Kubernetes.

3.4.3 Blueprint Deployment and Lifecycle Management

At the time of writing, the NFVCL supports three open-source cores: Free5GC, OpenAirInterface (OAI) and SDCore. Moreover, different deployment options are available for the UPF, namely, as a VM or a or a pod on a K8s cluster. For the sake of brevity, in the following we will analyse the creation process and the Day-2 operations at a general level, highlighting, if necessary, any operations that are specific for a certain core. Moreover, excluding the SBA, which is strictly deployed as a pod on a K8s cluster, the other components can be deployed as VMs or pods. The VM Provider and Helm Provider are the components created to manage VM and pod operations in the NFVCL. The following description accounts for a case in which gNB and UPF are deployed as VMs, but the flow charts reported in Annex A report the pod deployment as well. From the procedure standpoint, the steps are very similar; the implications for the performance are outlined in the following section.

As mentioned above, the NFVCL can automatically drive the creation of a core, the additional calls performed by the VM and Helm providers and related lifecycle operations, (hereinafter referred to as Day-0, Day-1 and Day-2, respectively). It is worth pointing out again that the following description represents completely zero-touch procedures that produce fully working configurations. The workflows reported in Annex A also show the number of generated code lines.

The **<u>core creation</u>** process orchestrates the deployment and configuration of essential 5G core components using a combination of blueprints and infrastructure providers through the interaction of blueprint modules, VM and Helm providers. Supported VM providers are OpenStack and Proxmox. This automated workflow ensures the seamless setup of such essential components, namely the SBA, the router and the UPF.

The operation begins with the GENERIC_CORE Blueprint, which initiates the creation sequence of the GENERIC_UPF Blueprint. In turn, the UPF Blueprint initiates the creation of the GENERIC_ROUTER Blueprint. These nested calls are also important during the core deletion phase, because blueprints store a hierarchical structure of their child blueprints, allowing for the correct deletion of each deployed component.

The VM provider manages the creation and configuration of the router VM, after which the router blueprint makes its details available via a callable function. Once the router is ready, the UPF blueprint proceeds to provision its own VM. To enable data routing through the network, the UPF blueprint interacts with the router blueprint to request the addition of routing information. The router is reconfigured accordingly, and confirmation is sent once the new routes are in place. Upon successful creation and configuration of the UPF and routing infrastructure, the UPF blueprint notifies the CORE blueprint, which then queries the UPF for its connection details. With this foundational network layer established, the CORE blueprint proceeds to deploy the core network components by instructing the Helm Provider to install the core Helm chart. This action represents the deployment of the main 5G core services.

Finally, the CORE blueprint coordinates with the GENERIC_GNB Blueprint to configure it. This involves the VM Provider once again, which handles the underlying configuration of the GNB VM. When configuration is complete, the GNB blueprint confirms readiness to the CORE blueprint.

This end-to-end process results in a fully provisioned and configured 5G core network, complete with UPF, router, gNB, and core services.

To **add a DNN**, the GENERIC_CORE updates its values according to the new DNN value, then send them to the Helm Provider to update the pods accordingly.

To **add a slice**, the GENERIC_CORE updates its values according to new slice value then sends them to the Helm Provider to update the pods. Once the pods are successfully updated, the GENERIC_UPF Blueprint must also be updated to align with the new configuration. However, before proceeding with the UPF update, a routing validation must be performed to ensure that the network paths are correctly established and consistent with the new slice configuration. Additional operations are required by some of the available cores. Namely, for Free5Gc is necessary to restart the SMF after each UPF reboot. This is otherwise the connection between the two components will not be stabilized. OpenAirInterface instead always requires a UPF restart when core data is changed, which causes pods to restart, for the same reason as Free5Gc.

The **addition of a new TAC** is initiated by the GENERIC_CORE with the creation of a new UPF instance. Then, the GENERIC_UPF creates a new router blueprint instance (router_5g). The GENERIC_ROUTER then contacts the VM Provider to create a new VM for the router. Once the VM is created, it is configured by the VM Provider.

Next, the GENERIC_UPF asks the VM Provider to create and configure a VM for the UPF component. Once this VM is ready, the GENERIC_UPF Blueprint requests the GENERIC_ROUTER to add routing rules to enable data routing. The router instructs the VM Provider to configure these routes, and confirmation is returned when the configuration is complete.

After the routing is set up, the GENERIC_UPF notifies the GENERIC_CORE that the UPF is ready. The GENERIC_CORE then retrieves the updated UPF information and uses this data to create new configuration values for the Helm Provider to update the core Helm chart. The Helm Provider applies the update and confirms that the values have been successfully updated.

Once the core configuration is up to date, the GENERIC_CORE initiates the configuration of a new GNB by coordinating with the GENERIC_GNB.

After configuration is completed, the GENERIC_CORE Blueprint acknowledges that the entire process, including the addition of the new TAC, has been successfully completed.

Adding a UE is different depending on the Core you're considering. SDCore also includes subscriber data among its values, so in its case, simply adding the new data and launching a pod update will suffice. Free5GC and OpenAirInterface, on the other hand, have the UDR that exposes the APIs needed to add a new subscriber. The process for **deleting a UE** is the same.

To **delete a TAC**, the GENERIC_CORE call the delete function on the GENERIC_UPF associate at that area. The GENERIC_UPF also calls the delete function on the GENERIC_ROUTER associate at that area. The VM_PROVIDER first deletes the GENERIC_ROUTER and then the GENERIC_UPF, after that GENERIC_CORE updates his configuration and sends it to HELM_PROVIDER to update pods.

This deletion operation follows the hierarchical structure of the blueprints, allowing for the correct deletion of each deployed component. The "final_cleanup" function is called as the last operation when deleting a core component and is used to delete the resources instantiated on the provider on which that component was running.

Deleting a slice begins with deleting it from the core values. It is then necessary to update the values of the UPF associated with that slice and the routing rules on router.

Deleting a DNN first involves removing it from the GENERIC_CORE values followed by updating the pods. The same procedures are applied for the **addition and deletion of a UE.**

The **core deletion** workflow is initiated by the GENERIC_CORE Blueprint that sends a request to the GENERIC_UPF Blueprint to delete its blueprint. The GENERIC_UPF Blueprint, in turn, triggers the deletion of the GENERIC_ROUTER Blueprint. Once the VM is destroyed, the router blueprint performs final cleanup, and the blueprint is removed. Next, the GENERIC_UPF Blueprint proceeds to request the VM Provider to destroy its own UPF VM. The VM Provider confirms the VM destruction, after which the UPF blueprint performs its final cleanup tasks and is marked as deleted. After the UPF has been removed, the GENERIC_CORE Blueprint coordinates with the **Helm Provider** to uninstall the core Helm chart. Finally, the GENERIC_CORE Blueprint performs its own final cleanup. At this point, all the associated components have been properly removed, and the entire network teardown process is complete.

Results

Several testing campaigns have been run to assess the performance of the NFVCL. Deciding how to carry out such an assessment is a non-trivial task: the obvious would be to compare the time required for deploying and performing lifecycle operations on a 5GS manually and in the presence of the NFVCL; however, it is very hard to compare an automated and a manual operation, as the former is somewhat deterministic while the latter heavily depends on the skills/speed of the operator. This is likely the main reason why a comparison of the time it takes to perform lifecycle operation in different open-source cores is not yet available in the state of the art.

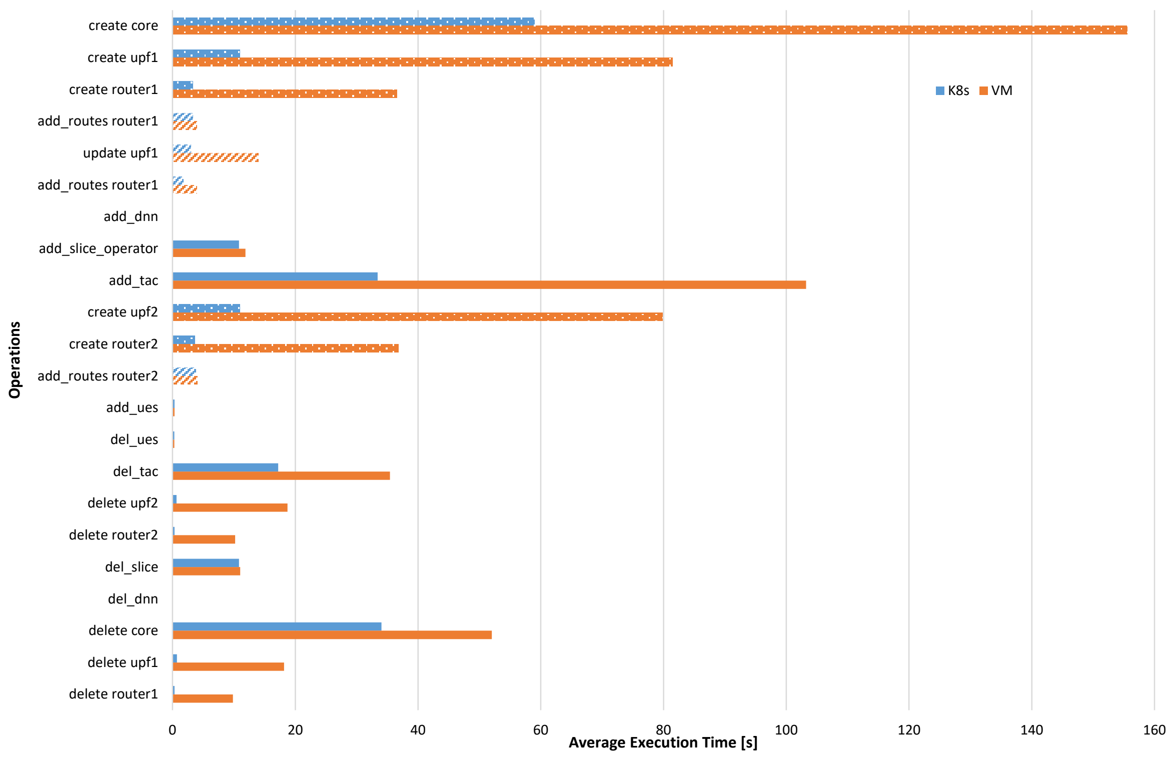


Figure 3-39. Execution time of lifecycle operations performed on Free5GC.

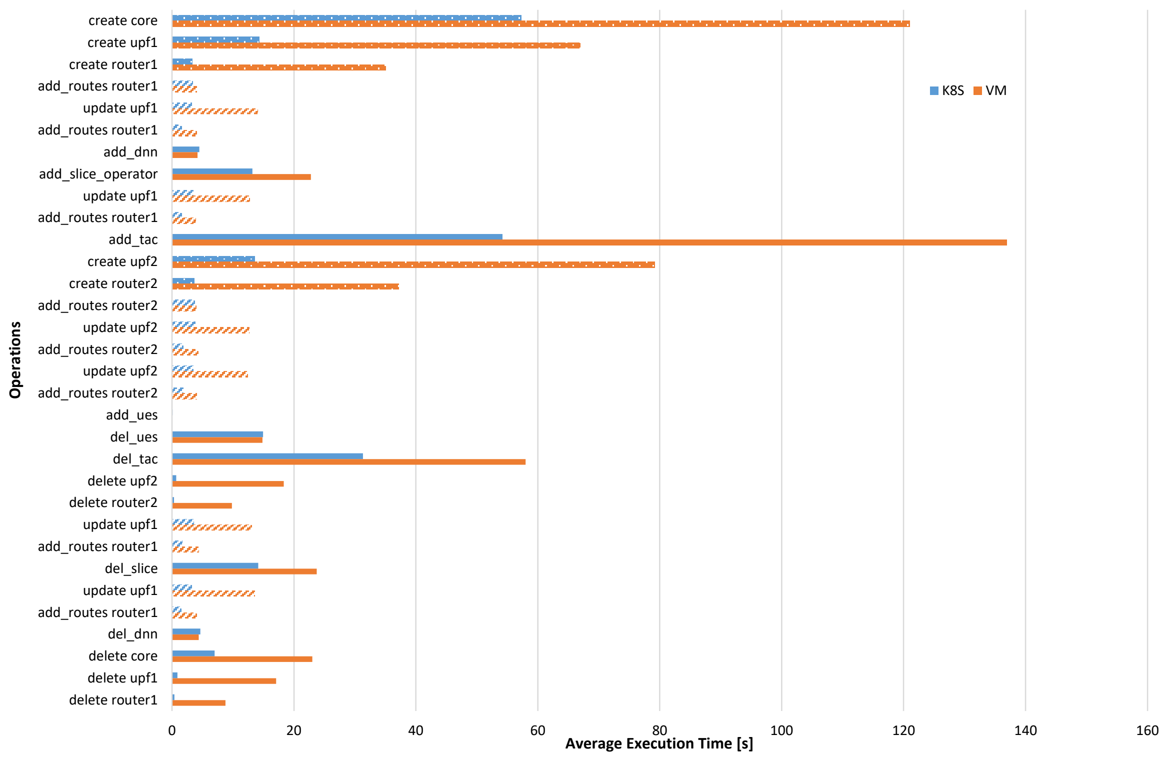


Figure 3-40. Execution time of lifecycle operations performed on OpenAirInterface.

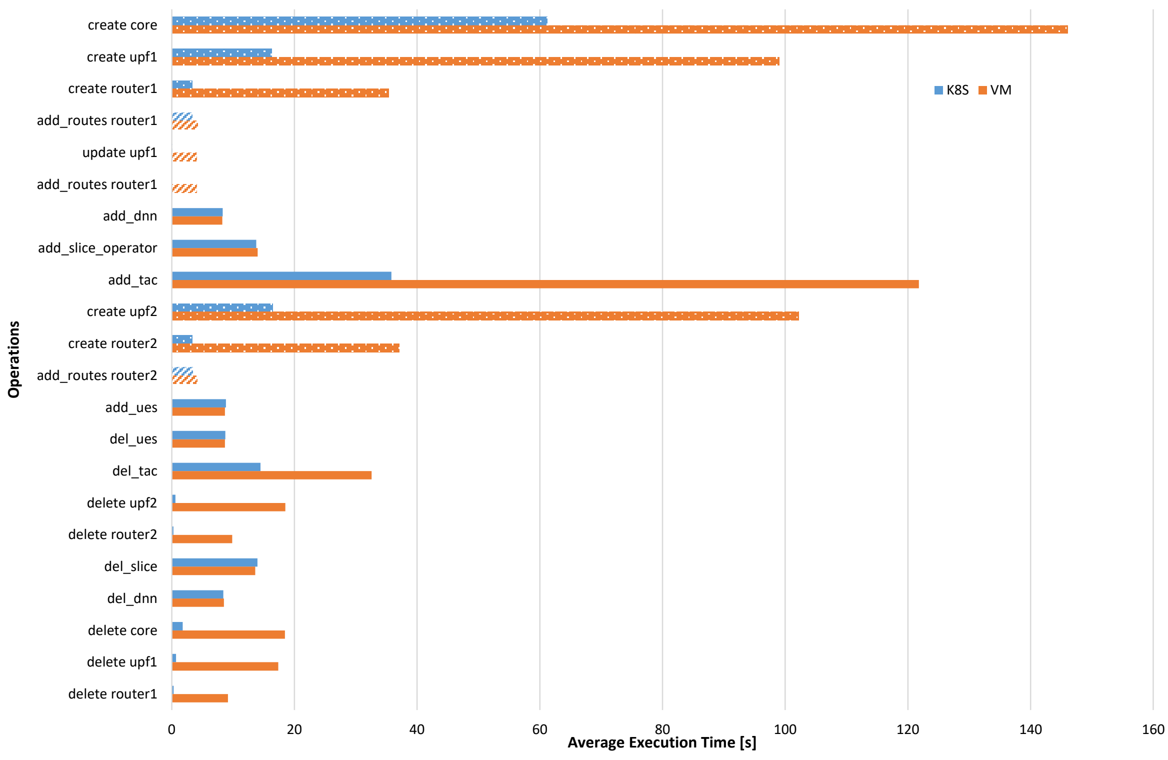


Figure 3-41. Execution time of lifecycle operations performed on SDCore.

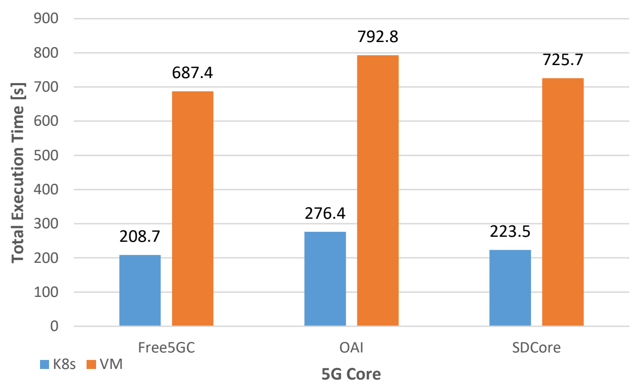


Figure 3-42. Sum of the execution times categorized in the above Figures.

In order to proceed with the assessment and, at the same time, contribute to the body of knowledge on upcoming 5G technologies, we decided to first provide a breakdown and a comparison of the time needed to manage the lifecycle of three well-known open-source cores, namely Free5GC, OpenAirInterface (OAI) and SDCore. Following, we provide a breakdown of the number of code lines automatically generated by the NFVCL to configure the 5GS at runtime to highlight the benefits of automation. Figure 3-39-Figure 3-41 show the average time (over 10 tests) that it takes to perform the lifecycle operations described earlier in this section. Day-0 operations are represented with a dotted pattern, Day-1 with diagonal stripes and Day-2 is solid. The first, most prominent feature is the higher time required by the cores that deploy the UPF in a VM, which is experienced for all the cores: indeed, the creation of VMs takes longer than spawning a pod on K8s. In order to minimize this overhead, the NFVCL allows to work on already available execution environments and VM images, however the hypervisors are responsible for the additional deployment time seen in these results.

It is also worth noting that OAI requires additional operations because its UPF contains information related to the slice, which means it has to be rebooted upon changes to the slices and TACs. OAI is also less "stable" than the other cores, hence sometimes the same operations need to be repeated in order to avoid misconfigurations. Free5GC also requires additional operations as the SMF needs to be restarted upon update of the UPF, but the resulting overhead is negligible and does not emerge from the results in Figure 3-39-Figure 3-41.

In total, VM deployments take three times as long, as shown in Figure 3-42. Although OAI requires more steps to perform the same lifecycle operations, the total time that it takes is just slightly higher with respect to Free5GC and SDCore. On the other hand, while Free5GC and SDCore performs the exact same operations, their execution times slightly vary: for example, the core creation and deletion take longer for Free5GC but other Day-2 operations (e.g., UE addition/deletion) take longer for SDCore because Free5GC offers specific APIs for UE management while SDCore requires pod reboot upon configuration updates.

Further considerations can be drawn by summing the execution times on a per-Day basis and showing the minimum and maximum values along with the averages, reported in Figure 3-43 (UPF deployed in a pod) and Figure 3-44 (UPF deployed in a VM). For both VM and pod deployments, Day-0 operations have the highest deviation from the average, especially for Free5GC. While the three core releases have very similar values in Figure 3-43, when the UPF is deployed in a VM OAI takes less time than Free5GC and SDCore, because the Docker setup is more complex and time-consuming, and Free5GC has a less deterministic Day-0 execution time.

Day-1 operations have quite similar minimum, average and maximum values for the three cores, and the differences among them are almost the same when the UPF is deployed in a pod or in a VM. The same can be said for Day-2 operations, with OAI taking slightly longer with respect to the other ones especially in the case of pod deployment.

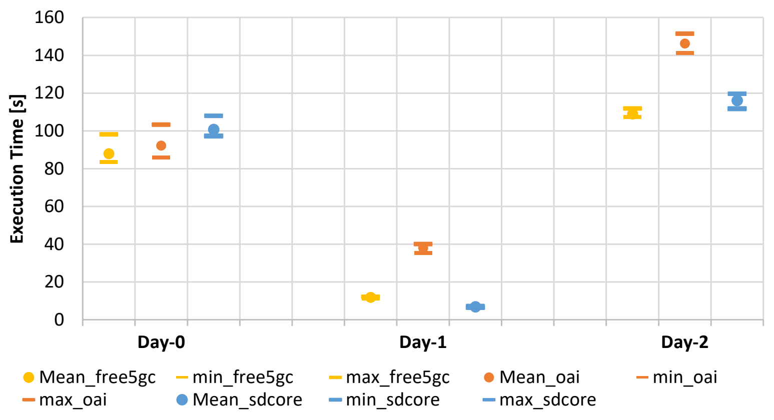


Figure 3-43: Distribution of the execution times for the three 5GSs with the UPF deployed in a pod.

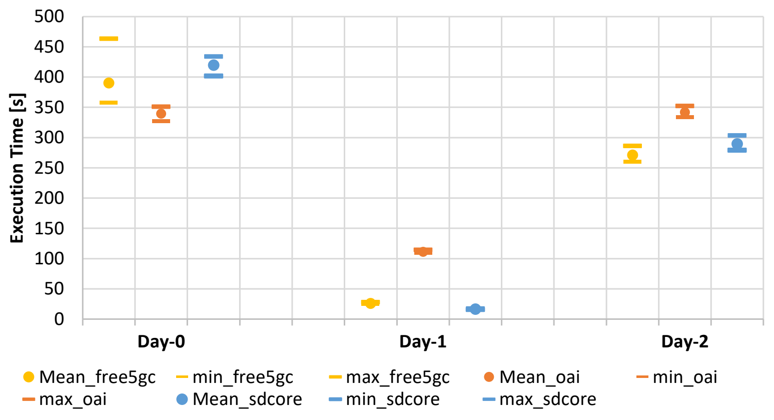


Figure 3-44: Distribution of the execution times for the three 5GSs with the UPF deployed in a VM.

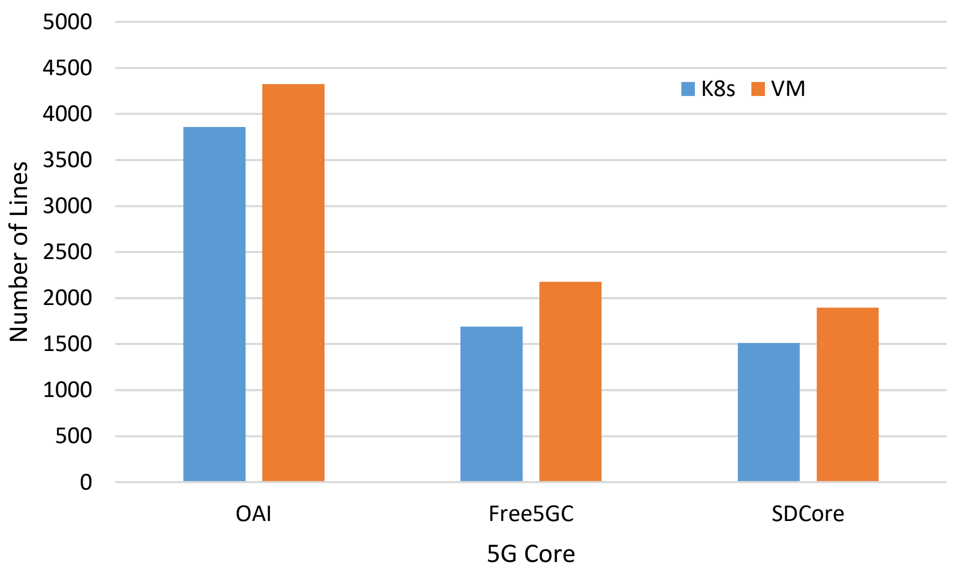


Figure 3-45: Number of code lines generated by the NFVCL for the automated configuration, deployment and orchestration of the three tested cores.

Finally, it is worth highlighting the most distinguishable feature of the NFVCL, namely automation. Figure 3-45 reports the number of code lines that are created in a zero-touch fashion for the three cores. The difference between a VM or pod deployment is around 500 lines for all cores, with Free5GC and SDCore presenting similar values while the OAI deployment requires the generation of more than double code lines, which is consistent with the higher number of required Day-1 operations. Automation of configurations is particularly useful during experimentation campaigns: along with the reduced times achieved by onboarding ready-for-use VMs, it allows experimenters to neglect the specificities of each core, reducing the time it takes for configuring the tests as well as the chance of errors. For instance, if we consider the lifecycle of a UPF deployed in a VM, the NFVCL allows to skip the VM creation, installation of the required, core-specific dependencies, Docker installation, image download and Docker compose editing. Moreover, it enhances the reproducibility of the tests, as the same configuration can be passed along and used with minor changes specific to their own execution environment (e.g., network names, topologies, etc.).

4 Network Slice Lifecycle and Power Management in Serverless Environments

4.1 Stateful FaaS for Energy Consumption Minimisation

4.1.1 Mathematical Modelling and Analysis

Serverless computing and the FaaS programming model are popular in the cloud [10] and they have attracted significant interest also at the edge [11]. With FaaS an application is made of a sequence of stateless function calls, which can be arranged in chains (i.e., f1 --> f2 --> … --> fN) or more complex structures, like DAG [12].

However, realistic applications typically do need function execution to be associated with some state, especially for edge applications, such as AI and real-time analytics [13].

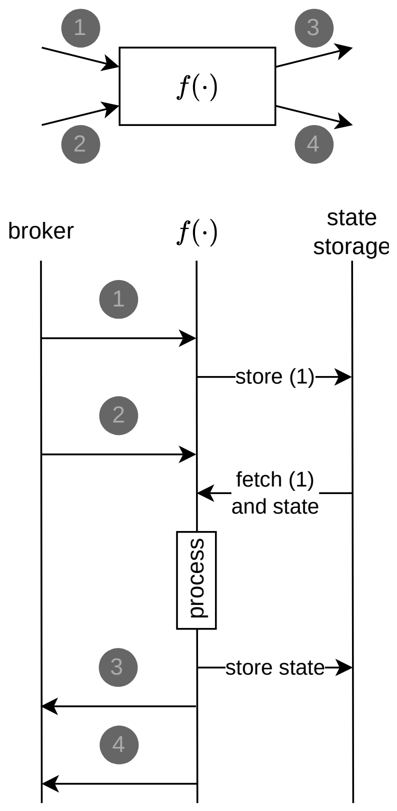


Figure 4-1: Example of how to realize stateful processing with stateless FaaS.

A straightforward solution to this problem, which we call *stateless FaaS*, is to maintain the state on an external storage system to be accessed on demand by the functions as part of their execution, as explained, e.g., in [14]. Such a deployment option is illustrated in the example in Figure 4-1, where function f(.) requires input from two dependencies (1 and 2) and has two outputs (3 and 4). When the function receives input 1, it is kept temporarily in the state storage. Once input 2 is received, full processing can occur combining the latter with the previous input 1 and the state, to produce the final outputs 3 and 4, after updating the state on the storage.

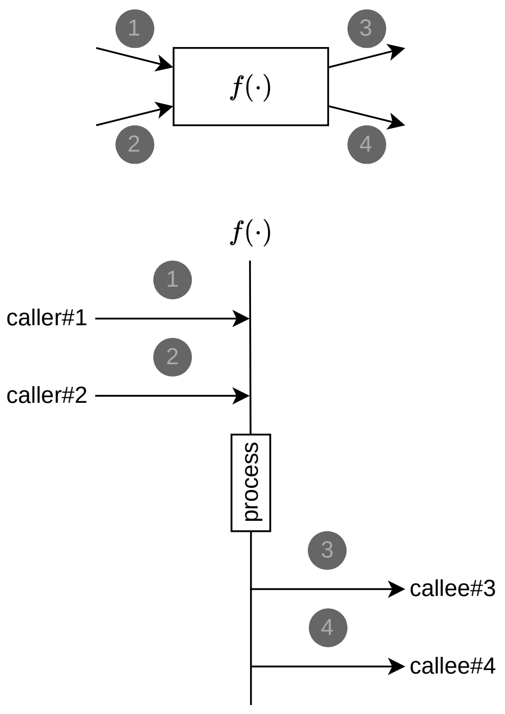


Figure 4-2: Example of stateful FaaS.

In common serverless computing platforms, function invocation happens through an HTTP command issued on a web server running in a container. Due to the lack of state, the same container can serve multiple users/sessions seamlessly, and the orchestration platform can easily perform autoscaling of such runners, i.e., decreasing or increasing the number of instances per function to match the instantaneous demand. An alternative to this strategy is dedicating each user/session to a runner, thus realizing what we call *stateful FaaS*. As illustrated in the example in Figure 4-2, with this model there is no need to fetch/update the state or store temporary input from previous function calls. In principle, the stateful FaaS model has two inconveniences. First, the number of runners may be much higher than that with stateless FaaS, because the former cannot exploit statistical multiplexing of multiple users/sessions like the latter. Second, if a runner is migrated from one node to another for any reason, e.g., system resource optimization, its internal state must be moved to the target host.

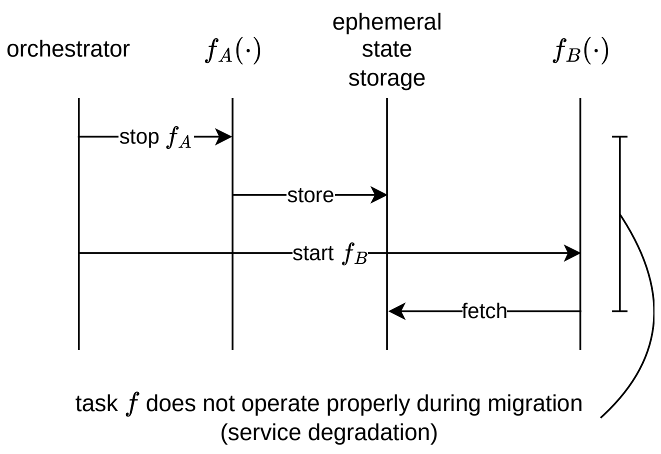


Figure 4-3: Migration of a stateful FaaS runner from node A to node B.

We show an example in Figure 4-3, where the orchestrator migrates a runner for the function f(.) from node A to node B. First, when stopping f(.) on node A the state is stored temporarily on an external system, which is then queried by the new instance of function f(.) on node B upon creation. With this solution, there would be a period during which the task performed by f(.) is not available. More sophisticated protocols can be devised [15], but, in any case, they would incur additional complexity or overhead, which is not needed with stateless FaaS. The impact of state migration on energy consumption is captured by the mathematical model defined and evaluated later.

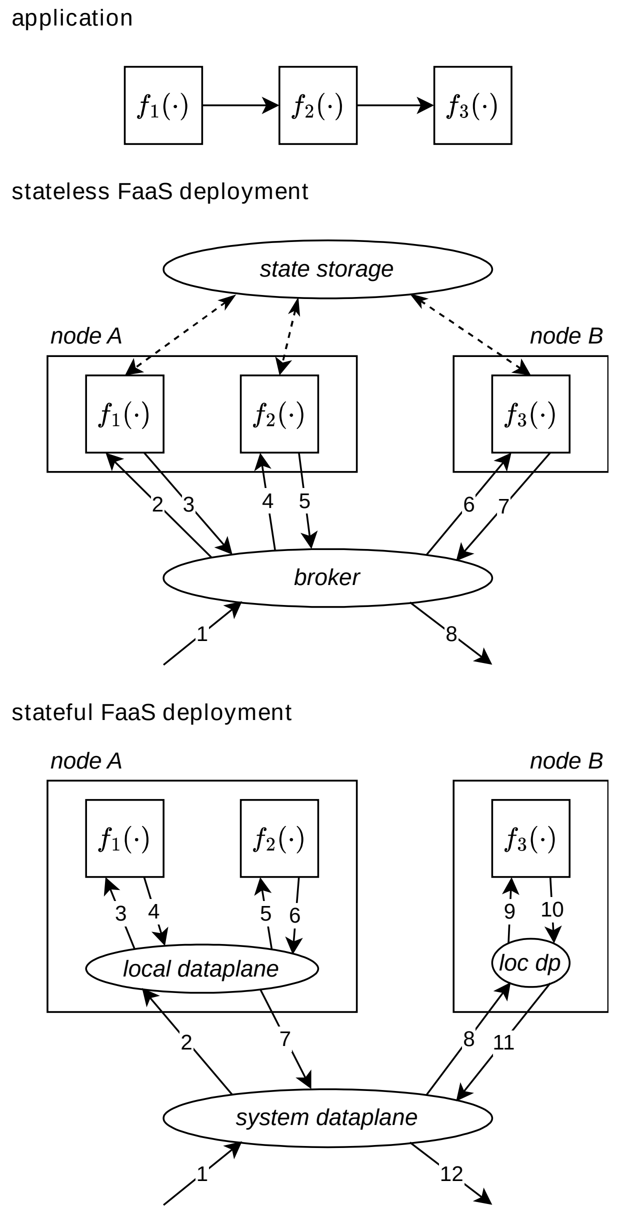


Figure 4-4: Deployment of a three-function chain (top) on two processing nodes through stateless FaaS (middle) and stateful FaaS (bottom).

We now illustrate deployment with stateless vs. stateful FaaS with the help of the example in Figure 4-4, with a three-function chain application running on two nodes A and B. In the example, we have one runner per function: node A hosts functions f1 and f2, and node B hosts function f3. With stateless FaaS, an intermediate layer is needed to dispatch function invocations to one of the matching runners: this is represented by a logical component called broker, borrowing the terminology from [16], which is an early study on the realization of distributed computing in pervasive systems. As can be seen, network traffic is generated at each function call for state access, on the state storage, and for invoking the next runner through the broker. On the other hand, with stateful FaaS, we need logical components to mesh the runners, which can be within a node or at a system level. Network access for accessing the state is unnecessary because the state is embedded within the runner. Furthermore, when a runner invokes another on the same node no network access is needed, too.

We now define a mathematical model to estimate the energy consumed in a time horizon T for executing the applications that enter/leave the system during that period. The model is intended to be used to evaluate high-level deployment strategies and run-time orchestration policies and, as such, it is not intended to provide quantitatively accurate results, but rather qualitative guidelines to drive algorithm design and high-level resource provisioning.

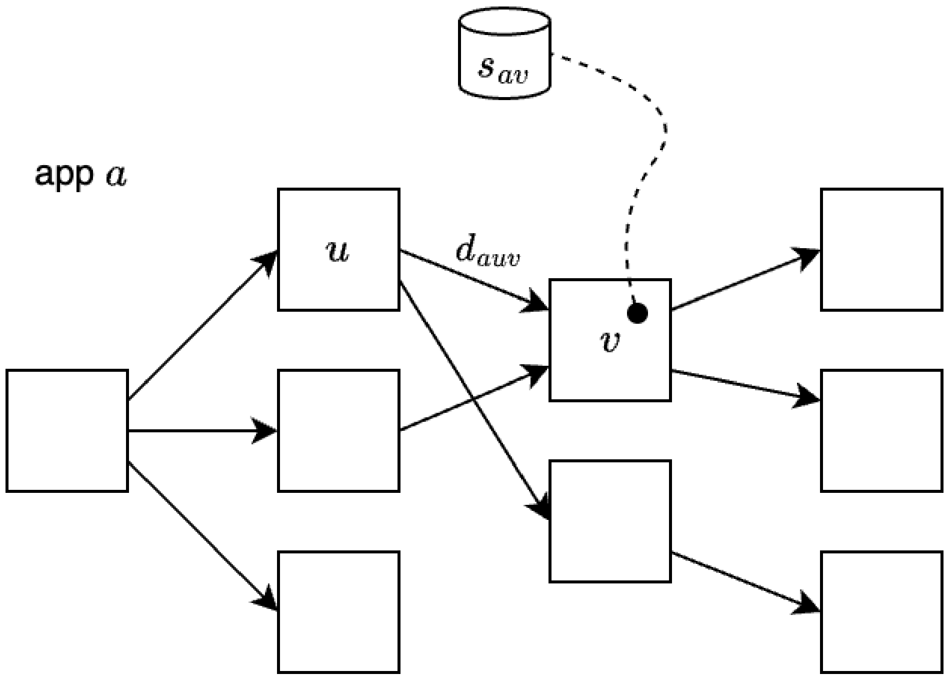


Figure 4-5: Application model. An app a consists of functions arranged in a graph. If function u calls function, v then an edge exists, and its weight d_{auv} is the amount of data exchanged. Each function v has a state of size s_{av}.

We assume the workload is made of applications (apps for short) that enter and leave the system dynamically at given times $t_a^{\downarrow}$ and $t_a^{\uparrow}$, for app a. An app a consists of some functions (or tasks) arranged in a directed dependency graph $G_a(V_a, E_a)$. Each vertex $v \in V_a$ is a task that depends on its predecessors (incoming edges) and produces output towards its successors (outgoing edges). The amount of data exchanged when task u calls its successor task v is d_{auv} , in bits. Without loss of generality, to have a more compact notation, we assume that the invocation rate is common for all the tasks within app a and equal to λ_a. Task v has an internal state of size s_{av}, in bits, and a processing request equal to r_{av}, in fractions of CPU. An example of a dependency graph is illustrated in Figure 4-5. In the following, we consider the system as dynamic, characterized by a series of discrete events happening at time $t_k \in \{t_1, \dots, t_N\}$, where t_N is the end of the period of interest and the other events correspond to an application entering or leaving the system. Between two consecutive events the power consumption remains stable (in a statistical sense) and we can characterize its average value through two step-wise functions, which are constant from time t_k until the next event t_{k+1} : $\alpha(t_k)$ is number of edge nodes used at time t_k to serve the active applications, where each node has a processing capacity C, in fractions of CPU; $\beta_a(t_k)$ is the average network traffic consumed by application a in the unit of time. We assume that the power consumption of an edge node is binary: if it is used, i.e., it serves at least one stateless FaaS or hosts at least one stateful FaaS runner, then it consumes a peak power; otherwise, if it is unused, it does not consume power at all.

Regardless of the deployment strategy, we can then define the total energy consumed in the system as follows:

$$E = \sum_{k=1}^{N-1} \Bigg[P_N \cdot \alpha(t_k) \\ + E_B \sum_{a \in A} \beta_a(t_k) \mathbb{I}(t_a^{\downarrow} \le t_k \le t_a^{\uparrow}) \Bigg] (t_{k+1} - t_k),$$

where P_N is the power consumption of an edge node and E_B is the per-bit network transfer energy, and $I(\cdot) \in \{0,1\}$ is an indicator function equal to 1 if and only if the condition is true. We focus on energy consumption assuming that there are no constraints on the availability of processing and network resources. In other words, we assume that the system can accommodate all the incoming requests, hence no admission control is needed. The notation used in the paper is summarized in Table 3.

Table 3: Notation used in the section. The last two rows are used only with Stateful FaaS.

Parameter	Description	Unit
$G_a(V_a, E_a)$	Task graph of app a. V_a is the set of tasks, E_a represents the invocation dependencies	
λ_a	Invocation rate of app a	s^{-1}
r_{av}	Processing request of task v at app a	CPU
s_{av}	State size of task v at app a	b
d_{auv}	Invocation data size from task u to v at app a	b
$t_a^{\downarrow}$	Arrival time of app a	s
$t_a^{\uparrow}$	Leaving time of app a	s
A	Set of all the applications in the period of interest	
$A(t_k)$	Set of applications active at time t_k	
$\alpha(t_k)$	Number of edge nodes active at time t_k	
$\beta_a(t_k)$	Traffic rate of app a at time t_k	b/s
P_N	Power consumption of a node	W
E_B	Per-bit network transfer energy	$\mu W/b/s$
E	Total energy consumed in the period of interest	J
C	Processing capacity of a node	CPU
t_k	Time instant of the k-th event	s
Δ	Defragmentation interval	s
$x_{av}(t_k)$	Mapping function indicating the index of the node to which task v of app a is allocated at time t_k	

For **stateless FaaS** we adopt a simple model that captures well its distinguishing features. Specifically, we assume that the number of active nodes needed at time t_k is the minimum possible, i.e.:

$$\alpha(t_k) = \left\lceil \frac{1}{C} \sum_{a \in A(t_k)} \sum_{v \in V_a} r_{av} \right\rceil$$

where $A(t_k)$ is set of applications active at time t_k. The inner summation is the total processing request of app a, which is then summed over all the applications and, finally, divided by the edge node capacity C. This implicitly assumes that no edge effects exist in horizontal scalability and the broker layer can distribute the load appropriately among the multiple task instances. On the other hand, the traffic rate of app a at time t_k is given by:

$$\beta_a(t_k) = \lambda_a \left[\sum_{v \in V_a} s_{av} + \sum_{(u,v) \in E_a} d_{uav} \right]$$

which is the sum of the traffic generated for the state access (first term) and function invocation between each node and its successors (second term), in the unit of time, as given by the invocation rate λ_a.

The model with **stateful FaaS** is more complicated because it depends on how tasks are assigned to edge nodes for three reasons. First, function invocation only consumes network resources if the two tasks are not assigned to the same edge. Second, since a stateful FaaS runner cannot be split/recombined, assigning the active tasks to available nodes to minimize the number of nodes used is akin to the bin-packing problem, which is known to be NP-complete. Finally, as active apps leave the system, fragmentation occurs (a term inspired by the similar effect in the memory management process of operating systems), i.e., edge nodes are only partially allocated: this is sub-optimal for energy consumption. To solve this problem, we foresee a defragmentation process to happen periodically, with the period equal to Δ, which is a system configuration parameter: during defragmentation, the active apps are rearranged to reduce the number of edge nodes needed, thus saving energy in the future. However, this process consumes energy because the state of some runners may have to be migrated from one node to another.

Now we introduce a last bit of notation: let $x_{av}(t_k)$ be a variable that indicates what edge node (using an arbitrary indexing scheme) hosts the runner for the task v of app a at time t_k. In time intervals where the app is inactive, i.e., before it enters or after it leaves the system, the variable is undefined. The values of $x_{av}(t_k)$ must be determined through two orchestration decision-making algorithms: i) when an app enters the system, the algorithm chooses where to deploy each of its tasks, by either selecting edge nodes already active (hosting other tasks) with sufficient residual capacity or activating new edge nodes; ii) upon defragmentation, the tasks of active applications can be migrated to other edge nodes to reduce the total number of the active ones. Determining an optimal policy for either of these decision processes has the same complexity as finding an optimal allocation for a bin-packing problem, as already mentioned. We propose to use the following simple heuristic based on the best-fit policy:

Stateful|best-fit algorithm:

- When an app enters the system, for each task we select the active node that hosts one of the predecessor tasks, if any (to save network traffic for function invocation). Otherwise, we select the active node that leaves the smallest residual capacity, if any, breaking ties arbitrarily. Otherwise, we deploy the task on an inactive node.
- Upon defragmentation, we apply the above algorithm policy to all the active apps, in arbitrary order.

We then derive the number of active nodes at time t_k as:

$$\alpha(t_k) = \left|\left\{x_{av}(t_k), \forall a \in A(t_k), \forall v \in V_a\right\}\right|$$

where $|\cdot|$ indicates the cardinality of the corresponding set, and the traffic rate of app a at time t_k is:

$$\beta_a(t_k) = \frac{1}{t_{k+1} - t_k} \sum_{v \in V_a} s_{av} \cdot \mathbb{I}\left(x_{av}(t_k) \neq x_{av}(t_{k-1})\right) + \lambda_a \sum_{(u,v) \in E_a} d_{auv} \cdot \mathbb{I}\left(x_{au}(t_k) \neq x_{av}(t_k)\right),$$

The first addend considers the state migration if the task was moved since the previous time event (by design, this can happen only during the defragmentation procedure) and the second addend considers the network traffic for function invocation, only if the task u and its successor v do not belong to the same node.

We conclude the section with the evaluation of the performance, in terms of energy consumption, of the stateless vs. stateful approaches, indicated as stateless|min-nodes and stateful|best-fit, respectively. For reference purposes, we also include two alternatives: stateless|max-balancing, as implied by the name, refers to a stateless FaaS system that seeks to maximize load balancing [17]; stateful|random is a variation of the stateful policy above, where there is no periodic defragmentation and the tasks of incoming apps are assigned to edge nodes at random, respecting the maximum capacity , and a new node is made active only if there is none with sufficient residual capacity. For full reproducibility of results, the source code of the simulator and the scripts and artifacts are available publicly as open source on GitHub[13].

The workload is created following the model in [18], which is inspired by real traces made available by Alibaba and broadly used in the literature, tuned as follows: the arrival and lifetime of apps follow a Poisson distribution, with average 1 s and 60 s, respectively; both the state size and the data invocation size are derived from the memory requirements produced by [18], by applying multiplicative factors called S (state) and D (data invocation), where D is always set to 100, which corresponds to the range [2, 303] kB, and S is expressed through the ratio S/D, which is 100 by default, in which case S would be in the range [0.2, 30.3] MB. The invocation rate is 5/s and the capacity of a node is set to 1000, which is sufficient to host any single task, whose requested capacity is drawn from an empiric distribution with a maximum value of 800. The edge node power consumption was set to 100 W, which is typical for a small device such as an Intel NUC; estimating the network consumption is much more complicated because it depends not only on the devices but also on the overall networking infrastructure: based on the results from a recent study [19], we have experimented with different values in the range [0.05, 5] µW/b/s. Each experiment lasted 1 day of simulated time and was repeated 1000 times; the plots show the average value across the repetitions with a symbol and the low (0.025) and high (0.975) quantiles as error bars. All the values above are to be considered unless specified otherwise.

In Figure 4-6 we show α and β[14] with different combinations of Δ and the S/D ratio, only with stateful|best-fit. β is affected significantly by both Δ and S/D: when the state is heavier ($S/D = 100$), the

[13] https://github.com/ccicconetti/stateful-faas-sim (experiment 001)

[14] We omit the subscript a as we plot the average traffic rate

network traffic is very high with small values of Δ (note the log scale on the y-axis) because frequent migrations are expensive. This effect is much less prominent with $S/D = 10$ and $S/D = 1$, because of the smaller state sizes compared to the invocation data sizes. With increasing Δ, all the curves initially decrease and then, increase again until they converge to the same value (as the defragmentation becomes more sporadic, the state size becomes less important). The minima of the curves depend on the specific value of $/D$. The number of active nodes is independent of S/D and always increases with Δ. The choice of Δ incurs a trade-off in the energy consumption of computation vs. network. In the following, we set the value of Δ to 120 s, i.e., twice the average app lifetime, which appears as a reasonable trade-off between network vs. processing consumption.

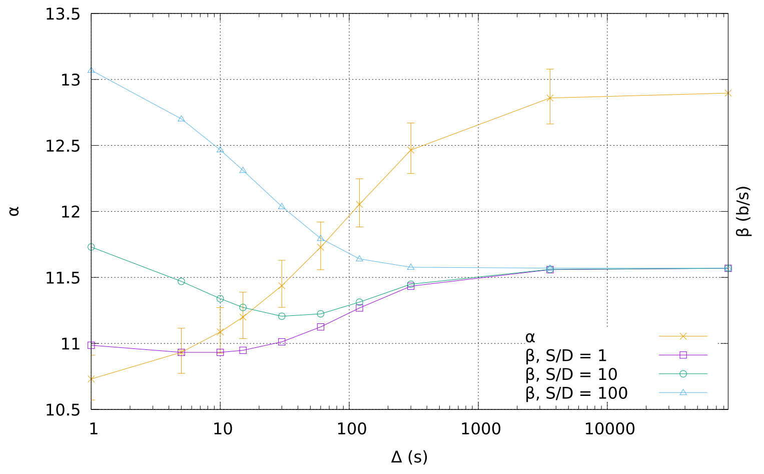


Figure 4-6: Simulations: α and β vs. defragmentation period Δ.

In Figure 4-7 we show the energy consumption with increasing E_B while keeping 100 W. The energy consumption increase with a higher per-bit-rate cost is higher with a stateless deployment, especially in the max-balancing flavour, and is very modest with a stateful deployment. In the latter case, we can see that the best-fit policy reduces energy consumption by about 2 compared to random, for all values of E_B. In the following, we only consider the two extremes of the E_B range.

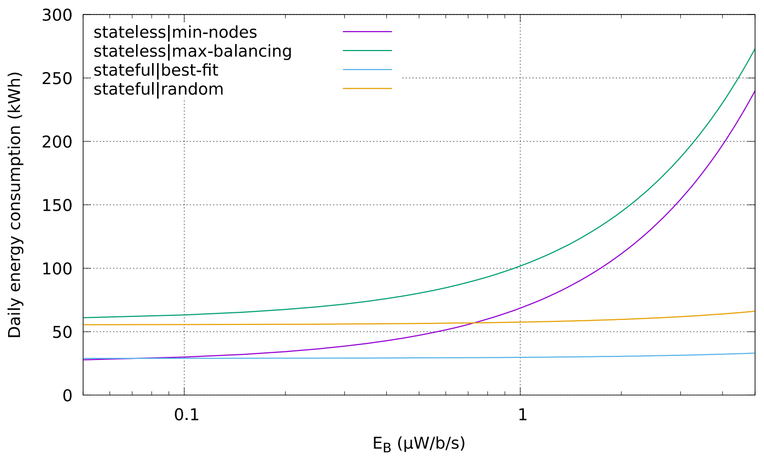


Figure 4-7: Simulations: energy consumption vs. E_B.

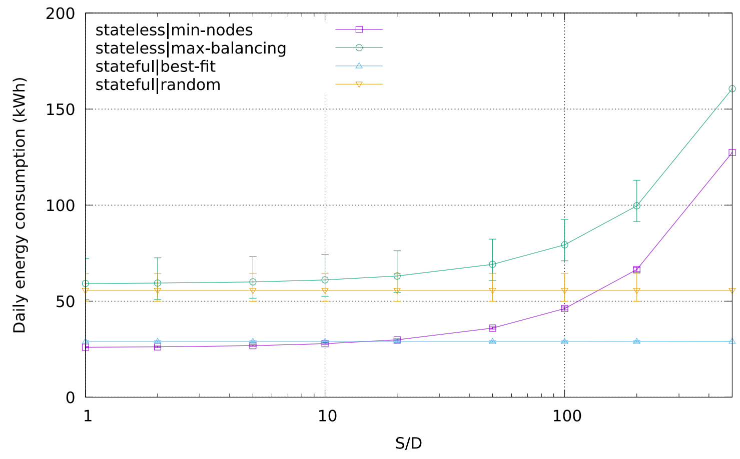


Figure 4-8: Simulations: energy consumption vs. S/D, $E_B = 0.05$ μW/b/s.

The impact of the state size, compared to the data invocation size, is exposed in Figure 4-8, with low per-bit-rate energy cost, i.e., $E_B = 0.05$ μW/b/s. A stateless deployment, with a min-nodes policy, is the best option only for $S/D \leq 10$ and only by a small margin compared to stateful|best-fit. On the other hand, as S/D increases significantly above 10, stateless deployment becomes significantly more energy-hungry, due to the cost of accessing the state upon each function invocation. With $S/D > 100$, stateless is outperformed even by stateful|random. The max-balancing policy follows the same trend as min-nodes and is always above the latter, though the gap reduces slightly as S/D increases. From an energy consumption perspective, stateful deployments are almost insensitive to the size of the applications' states.

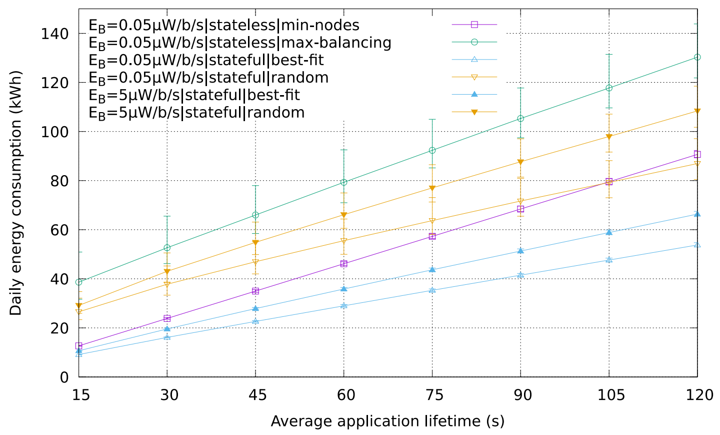


Figure 4-9: Simulations: energy consumption vs. average application lifetime.

In Figure 4-9 we report the measurements obtained with min/max E_B values for stateful policies (with stateless, the values with maximum E_B are well above the plot -axis range) when increasing the application lifetime from 15 s to 120 s. As expected, all the curves increase with the load. Both stateful|best-fit curves lie at the bottom and gain an increasing margin compared to all the others as the load increases. The second-best option is stateless|min-nodes (only with minimum E_B), while the stateless|max-balancing performs worst.

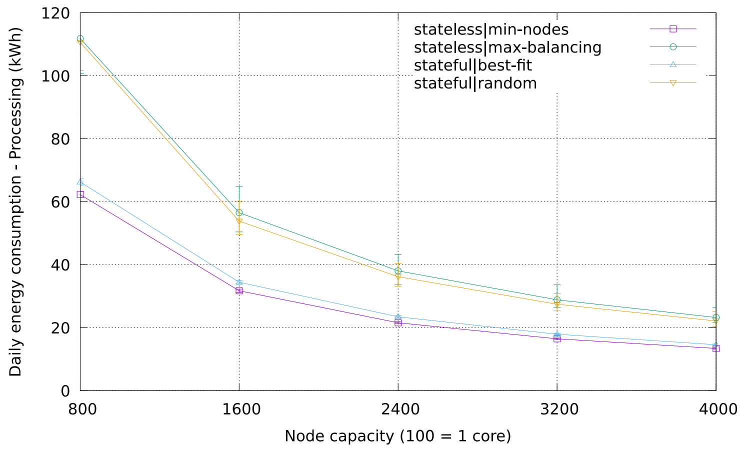


Figure 4-10: Simulations: energy consumption vs. node capacity.

Finally, in Figure 4-10 we show the energy consumption (only due to processing) with increasing node capacity from 800 to 4000. All the curves decrease because with increasing C the number of nodes required decreases, as well, while we keep the power consumption per node P_N constant. It is interesting to note that the curves are almost overlapping in pairs. At the bottom (less energy consumed) we find stateful|best-fit and stateless|min-nodes: in fact, they both aim at reducing the edge computing infrastructure energy consumption. Stateless has a slight gain compared to stateful, but it is more than compensated by a lower energy efficiency from the network traffic perspective. At the top (more energy consumed), the two comparison systems show similar performance, which can be explained by the fact that they both try to spread as much as possible the load among the active nodes: stateless|max-balancing does this explicitly, stateful|random implicitly. <u>A stateful deployment, with a best-fit allocation strategy, can be as efficient as a stateless one despite the fragmentation issue</u>.

4.1.2 Experimental Evaluation

We now illustrate the results obtained with a testbed of small edge nodes, related to the practical comparison of the stateless vs. stateful serverless computing paradigms.

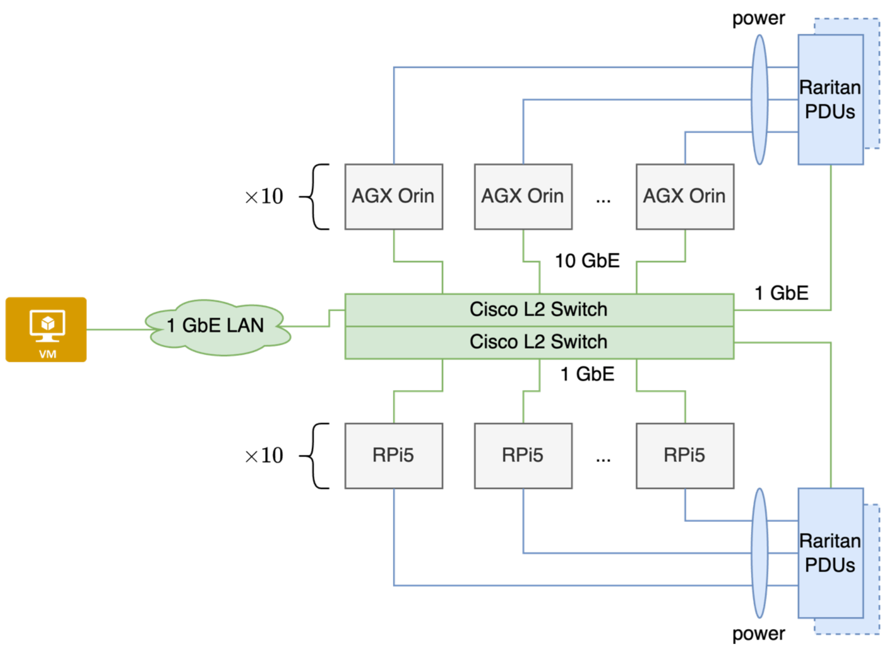


Figure 4-11: Testbed used for the evaluation of stateless vs. stateful serverless computing.

The testbed is hosted by CNR-IIT and is illustrated in Figure 4.11. It includes 21 hosts in total:

- 1 Virtual Machine running on an Intel server in the CNR data centre, interconnected with the other hosts via a 1 GbE LAN.
- 10 NVIDIA AGX Orin 64 Gb embedded devices.
- 10 Raspberry PI 5 single-board computers.

The following ancillary devices were used for the experiments:

- Cisco L2 switches, in stack mode, providing all the hosts with 1 GbE (RPi) and 10 GbE (Orin) connectivity.Raritan PDUs providing the hosts with power and monitoring the active power of each individual device.

The experiments have been executed with the EDGELESS[15], which is a platform that allows the development and deployment of stateful agents in the edge-cloud. A single cluster was configured including all the hosts, managed by a single orchestrator running on the VM. The scripts to run the experiments and to analyse the data are all available publicly, together with the artifacts of our experiments, on a GitHub repository[16].

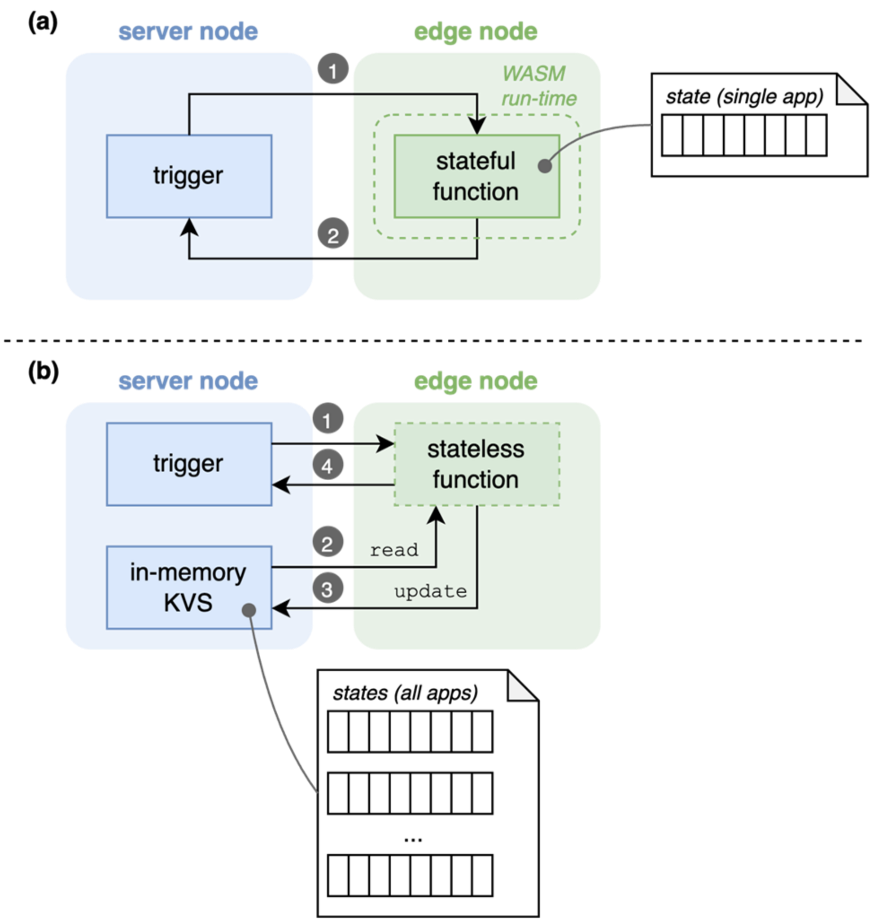


Figure 4-12: Workflows used for the experiments: (a) stateful, vs. (b) stateless.

[15] https://github.com/edgeless-project/edgeless/
[16] 009-6green-state, 010-6green-calib, and 011-multicore

In Figure 4.12 we illustrate the two workflows (applications) used for the experiments. In (a) the workflow consists of a trigger function that generates messages with Poisson-distributed interarrival times. The message is sent to a stateful function that performs a processing operation on its internal state. In particular, the state consists of a vector of 32-bit floating point numbers, initialized with random values between 0 and 1, and the operation is the element-wise computation of the trigonometric sin() function. After the operation is complete, a message is generated towards the trigger function to record message latencies. The workflow in (b) is functionally equivalent but the state, i.e., the vector, of each application is kept in an in-memory Key Value Store (KVS) hosted on the server VM. Therefore, the stateless function is forced to read the vector before each operation and update it with the new values afterwards. The workflows were configured with annotations that forced the orchestrator to assign the trigger function instances to the VM, while the stateful/stateless function instances to the edge nodes in a random fashion.

Calibration experiments. We have run initial experiments to calibrate the system parameters, whose results are reported in the following.

First, we have created an incremental number of stateful workflows, one every 60 seconds, deployed on the same Orin. Each workflow had a rate of 80 Hz, with a state of size 100k (i.e., the vector had 100k elements, corresponding to an in-memory size of 400 kbytes).

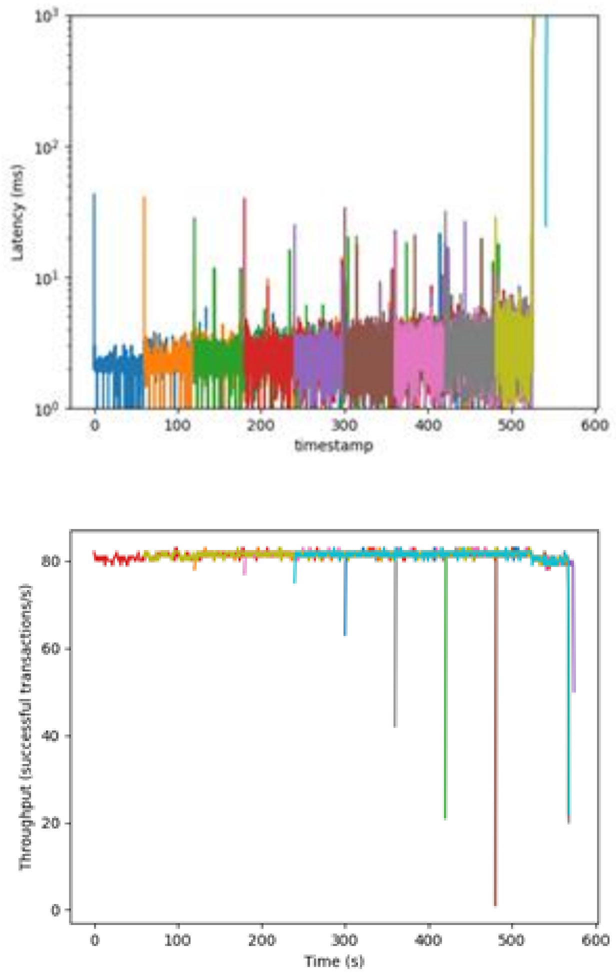


Figure 4-13: Calibration experiment with increasing stateful flows. Left: workflow latency. Right: Throughput.

In Figure 4.13 (left) we show the workflow latency over time, which increases only slightly until the node becomes overloaded after the 10-th workflow is added. Similarly, in the right part we can see that the throughput of the workflows is stable until the last flow, with spurious spikes only occurring whenever a new flow is added for edge effects in post-processing the data.

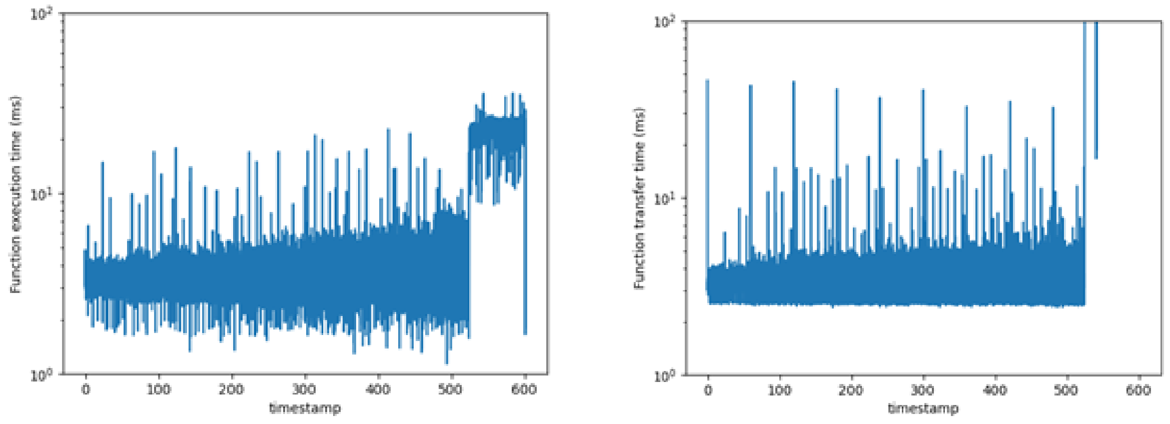


Figure 4-14: Calibration experiment with increasing stateful flows. Function execution (left) vs. transfer (right) time.

In Figure 4.14 we break down the workflow latency in the two main components, which are the time needed for the processing operation (left) and the latency introduced by the network and trigger function (right), called function transfer time. The latter has a more stable behaviour than the former, with spikes that are caused by the initialization of the state when a new workflow is created. When the system becomes unstable, after the last workflow is added, the function execution time remains bounded, but the transfer time grows indefinitely.

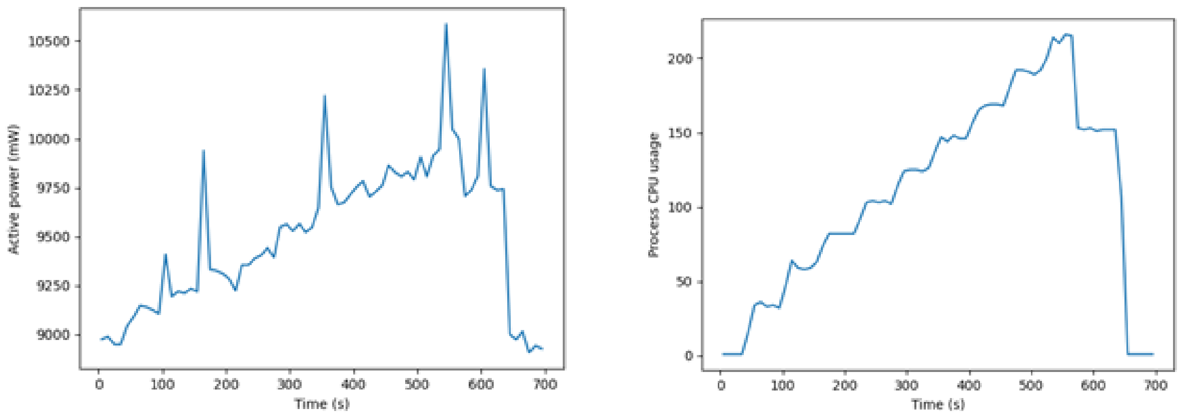


Figure 4-15: Calibration experiment with increasing stateful flows. Left: active power. Right: CPU usage.

Finally, in Figure 4.15 we report the active power (left) and CPU usage (right). It is interesting to note that there is a clear positive correlation between these metrics, whose values follow the same qualitative trend. This confirms the intuition that the active power of AGX Orin devices is a linear function of the CPU usage, with an offset given by the idle consumption.

We now report the results from a second batch of calibration experiments. We used both Orin and RPi devices, but always one at time for each experiment. Again, we only deployed stateful workflows with a message rate of 80 Hz. We repeated multiple experiments, with 1 vs. 10 workflows, and with variable state sizes from 1k to 1M elements.

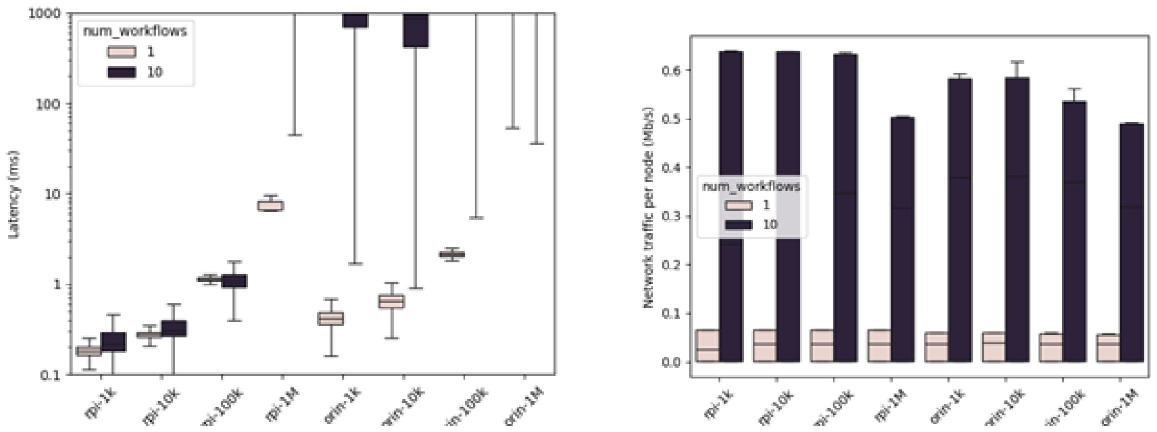


Figure 4-16: Calibration experiments with various state sizes. Left: workflow latency. Right: network traffic.

In Figure 4-16 (left) we report the workflow latency. As can be seen, the latency increases with the state size, because more sin() operations are needed. The RPi 5 device can withstand greater state sizes than the AGX Orin, which can seem counterintuitive because the latter is more powerful. However, the latter has 8 CPU cores, while the RPi 5 has only 4, therefore the per-CPU processing power of the RPi 5 is greater. We note that, in EDGELESS, function instances execute in a WebAssembly run-time environment that is designed for single-thread operation. In the right plot, we report the network traffic per node, which takes into account the messages exchanged in the data plane, as well as the control and management of the EDGELESS node services. The traffic is not affected by the state size, because the functions are stateful. There is a light decrease only when the system is unstable.

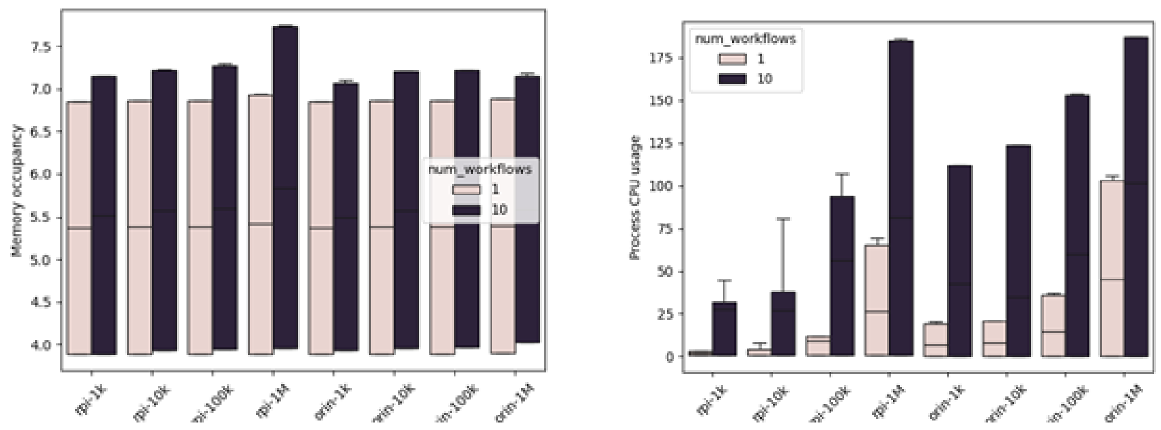


Figure 4-17: Calibration experiments with various state sizes. Left: active power. Right: CPU usage.

Finally, in Figure 4-17 (left) we report the memory occupancy of the EDGELESS service running in the nodes, in percentage of the overall memory available. As expected, the occupancy increases with the state size, but the increase is modest compared to the baseline, because of the relatively small size of the state footprint in memory; even with 1 M elements, each state occupies 4 MB, while the RPi 5 is equipped with 8 GB of RAM and the AGX Orin with 64 GB, shared between CPU/GPU. The right plot shows the CPU usage, which on the other hand increases significantly with both the state size and the number of workflows, because of the CPU-bound nature of the application used in the experiments.

Full experiments. In the full experiments 20 devices (10 RPi + 10 Orin), with 20 and 200 workflows, state size of 10, 1k, and 100k elements, and we compared the two patterns stateful vs. stateless. In particular, for the stateful case we deployed precisely the given number of workflows (20 or 200), each with a message rate of 100 Hz, where function instances are assigned at random to nodes by the orchestrator. On the other hand, to mimic a typical serverless computing deployment, for the stateless case we forced the orchestrator to deploy exactly one workflow on each node, then we adjusted the message rate to emulate the same load as with a stateful workflow.

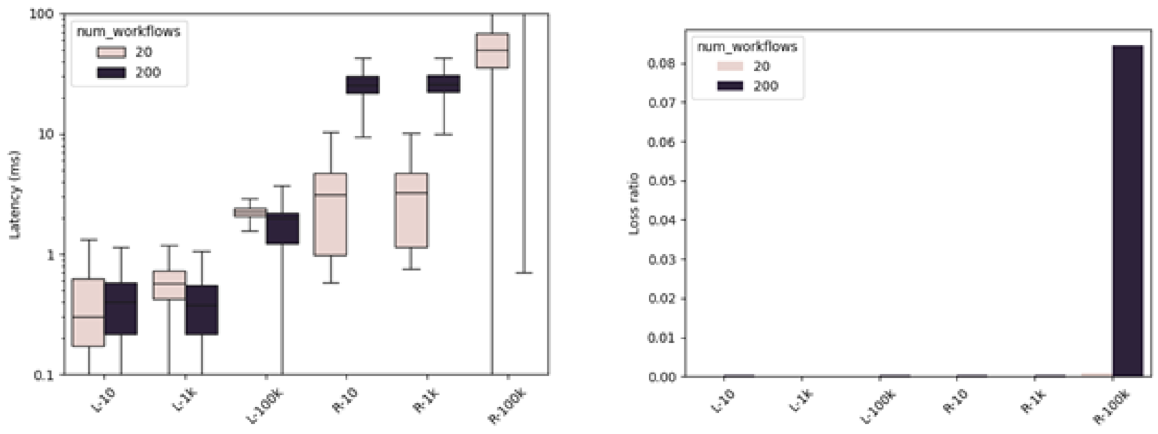


Figure 4-18: Full experiments, stateful vs. stateless. Left: latency. Right: loss ratio.

In Figure 4-18 (left) we show the workflow latency. The results are grouped based on the number of workflows, 20 or 200, which means an average of 1 or 10 function instances per node; remember that the assignment of function instances to nodes is done by an orchestration function at random, there it can happen that some node is loaded more than others. On the x-axis, we indicate labels that specify the deployment mode, i.e., stateful (L = local state) or stateless (R = remote state), and the state size, from 10 elements to 100k elements. The same format is adopted throughout the analysis. The plot leads us to several observations, which are confirmed by results shown later:

- Stateful deployment exhibits a significantly lower latency, not only with a large state (e.g., 100k), but also with a very small state of 10 elements. This is due to the cost of accessing the remote state, even if the latter is stored in a service in the same LAN as the edge nodes, which is an optimistic scenario. A more realistic would involve the state located in some cloud-hosted storage service, which would increase the remote access penalty in terms of latency.
- With a stateful deployment, the latency with 200 workflows is not significantly higher than that with 20 workflows. Rather, with 10 and 1k elements, which have a modest processing cost, the latency is slightly lower on average, and with comparable spread. Only with 100k elements the tail latency increases significantly (note the plot has a log-scale in the y-axis), but the average is still similar. This counterintuitive behaviour suggests that the nodes are not overloaded.
- However, with a stateless deployment, the latency with 200 workflows is significantly higher than that with 20 workflows, by at least an order of magnitude. Since this cannot be due to the processing

in function instances, which is the same as in the stateful case, we believe the effect is due to the contention on the state retrieve/update operations. In fact, when the state is greatest, i.e., with 100k elements, the system becomes unstable, with latencies growing arbitrarily. Figure 4-18 (right) confirms this by reporting the loss ratio, i.e., the ratio between the messages received back by the trigger function (see Figure 4-12) and those emitted by it: stateless with 200 workflows and 100k elements is the only case with a non-negligible loss ratio.

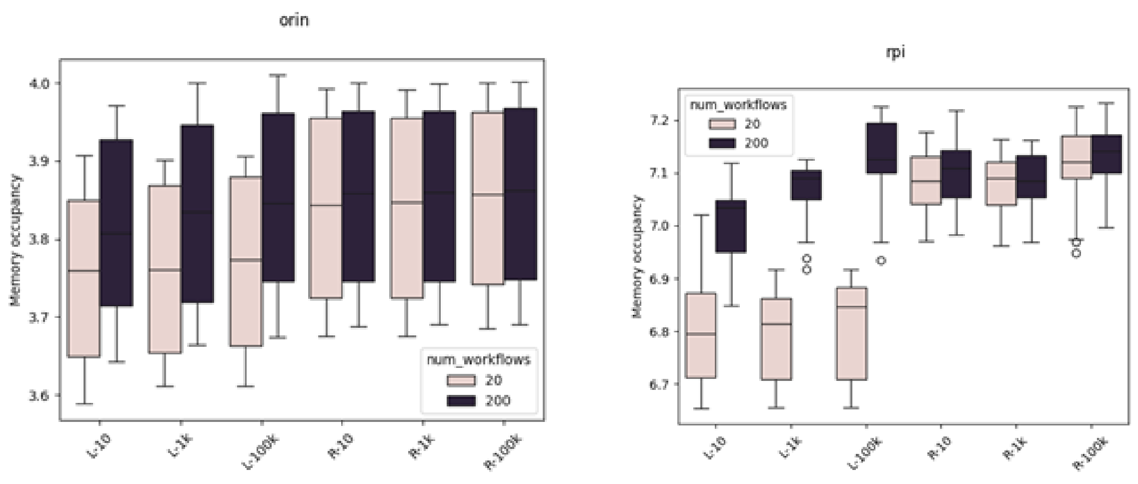


Figure 4-19: Full experiments, stateful vs. stateless. Memory occupancy of Orin (left) and RPi (right) devices.

In Figure 4-19 we report the memory occupancy, grouped by node type: AGX Orin devices on the left, RPi 5 devices on the right. Since the occupancy is expressed in percentage, the baselines for the two devices are different (AGX Orin ones have 64 GB of RAM, RPi 5 devices only 8 GB), but the qualitative behaviour is the same. With a stateful deployment the memory occupancy increases with the state size, which is fully expected because the state is kept locally at each function instance; on the other hand, a remote deployment is independent from the size of the state, which is stored externally. In our experiments, the state has a modest size compared to the availability, therefore the different memory occupancy difference is barely noticeable, in the order of 0.1%-0.2%. However the memory requirement of a stateful deployment may become a limiting factor when the state is either very large or the memory availability on edge nodes is severely constrained.

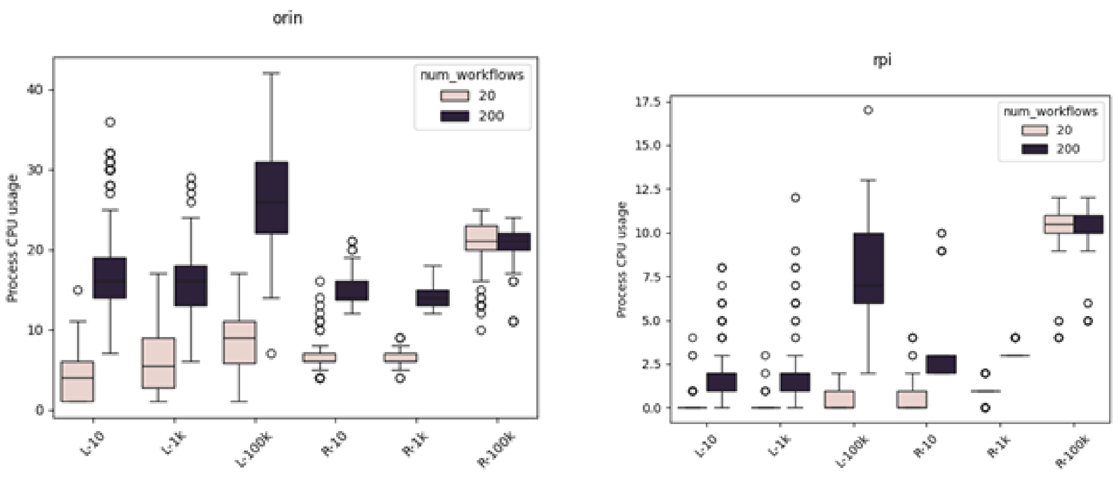


Figure 4-20: Full experiments, stateful vs. stateless. CPU usage of Orin (left) and RPi (right) devices.

In Figure 4-20 we report the CPU usage. The results confirm our observations about the latency. In fact, we see that, in general, the CPU is always underloaded. In relative terms, there is a significant increase from 20 to 200 workflows, due to the extra work; also, the CPU usage increases with the state size, because more sin() operations are needed. With a stateless deployment and largest state size, i.e., R-100k in the plots, the CPU usage is the same for 20 and 200 workflows only because the system is unstable: the service rate is not CPU-bound but rather state-access-bound.

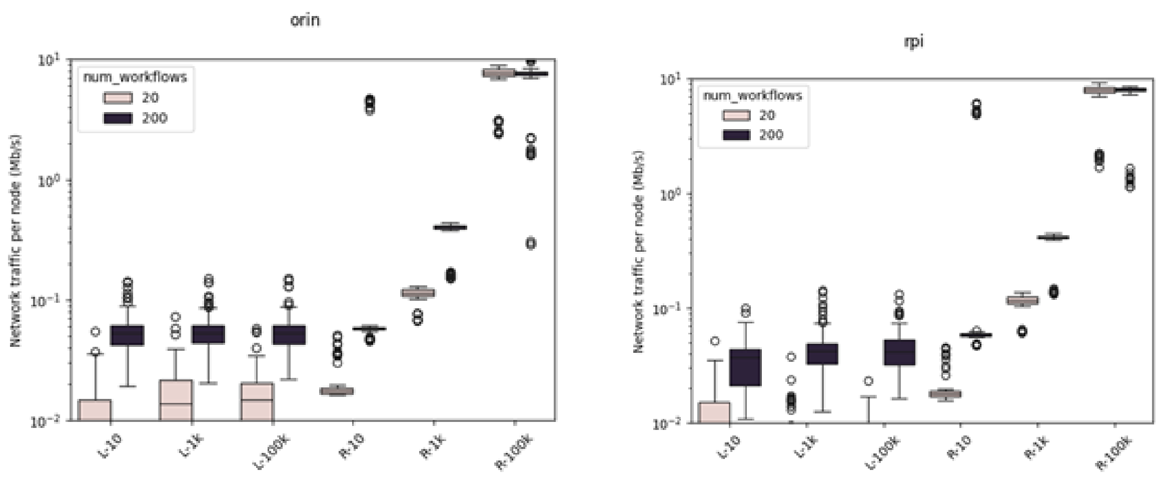


Figure 4-21: Full experiments, stateful vs. stateless. Network traffic of Orin (left) and RPi (right) devices.

In Figure 4-21 we report the average network traffic per node during the experiment. The results are comparable between AGX Orin a RPi 5 devices because this metric only depends on the amount of data required by the workflow, including state read/update operations with a stateless deployment, and for control/management plane signalling. The network traffic is minimum with a stateful deployment, where it depends only on the number of workflows but not the state size. Instead, it grows significantly with a stateless deployment because of the state-related network operations.

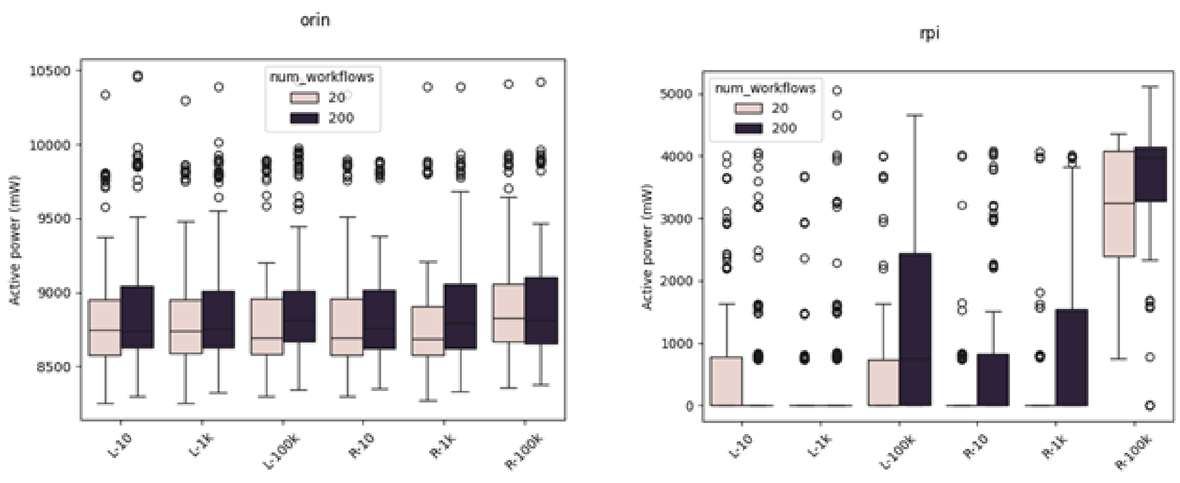


Figure 4-22: Full experiments, stateful vs. stateless. Active power of Orin (left) and RPi (right) devices.

In Figure 4-22 we report the active power. A general observation is that AGX Orin devices have much more stable power readings, while the RPi 5 devices exhibit erratic measurements. We speculate that this could be due to DVFS, and other power consumption mechanisms, performed by the RPi 5. Note that both categories of edge nodes have not been tuned for reduced power consumption and are using out-of-the-box configurations. Another general observation is that AGX Orin devices have a much higher baseline power consumption than RPi 5 devices, 8.5 W vs. 2.5 W, as confirmed by empirical evidence found in web forums. The results do not exhibit strong correlations of the active power with the deployment model, state size, or number of workflows, except for stateless deployment with 200 workflows on RPi 5 devices (labels R-10, R-1k, and R-100k in the right plot).

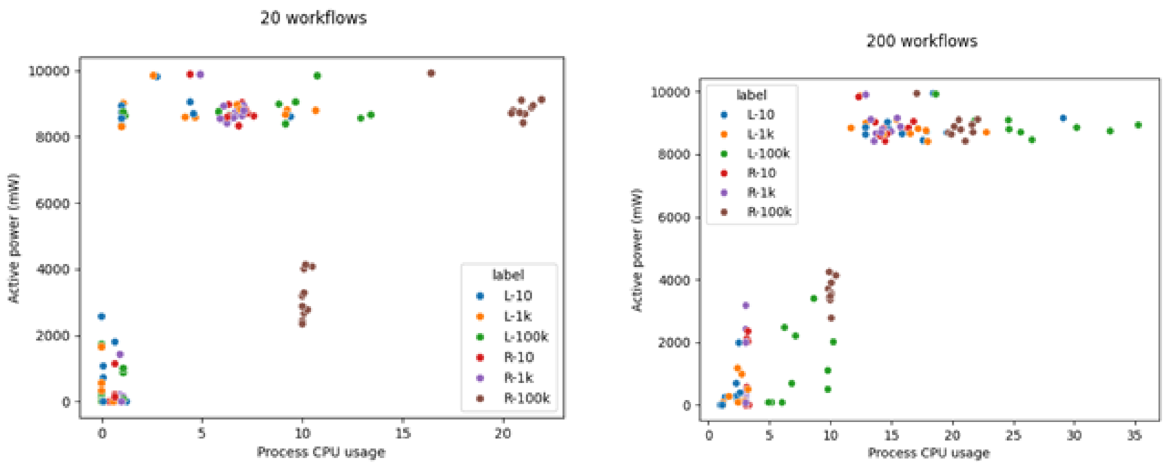


Figure 4-23: Full experiments, stateful vs. stateless. Active power vs. CPU usage with 20 (left) and 200 (right) workflows.

To delve deeper on this matter, we have broken down the active power results per node, instead of aggregating the data samples in box plots as reported in Figure 4-22. Figure 4-23 shows the average active power of each node for a corresponding value of average CPU usage, using different colours for the deployment model and state size, as well as grouping the results for 20 (left) and 200 (right) workflows. With 20 workflows (left plot) we can see about half of the points laying in a straight line, which suggests proportionality between the active power and the CPU usage: those points correspond to RPi 5 devices, as can be inferred by active power being lower than 5 W. On the other hand, AGX Orin devices (above 8.5 W) are basically independent from the experiment characteristics. With 200 workflows (right plot), the AGX Orin devices remain independent, while the correlation with RPi 5 devices becomes less evident. Comparing these results with the previous ones suggests exercising caution about the use of CPU usage as a direct indicator of power consumption with AGX Orin and RPi 5 devices, irrespective of the deployment model and overall load.

4.1.3 Conclusions

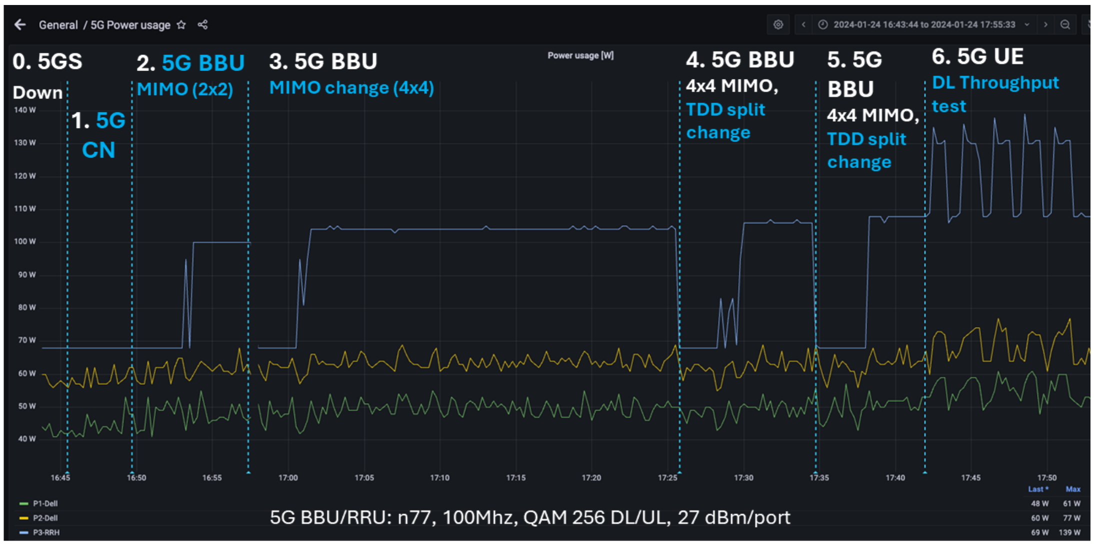
We have performed a comparative analysis of two deployment models for serverless workflows: stateless, which is the state-of-the-art approach where the application's state is stored at an external service and must be retrieved/updated when needed, and stateful, where a function instance is deployed for every application and, thus, can keep its state local. For the analysis, we have used simulation, based on a custom mathematical model, and testbed evaluation with 20 mixed edge nodes, i.e., Raspberry Pi 5 and AGX Orin devices. The simulation has led us to identify some key performance trade-offs, especially in terms of power consumption, depending on the state size and network characteristics. In brief, using a stateless deployment model is never the best choice, unless the state is very small or the external storage service can be accessed with negligible performance penalty. The testbed evaluation made this conclusion even stronger. In fact, despite the optimistic environment for what concerns the access to the storage service (an in-memory KVS in the same LAN as the edge nodes), a stateful deployment exhibited less latency and no noticeable degradation in terms of power consumption. We note that there might be cases when a stateful deployment is not possible for practical reasons, including insufficient memory availability on the edge nodes to keep the state, administrative requirements on the application state location (e.g., to comply with GDPR rules), or backward compatibility with a legacy codebase relying on a stateless deployment model. Finally, we have observed some correlation between the CPU usage reported by the edge nodes and their power consumption, as measured by a monitored PDU, but only on come conditions. Therefore, it is not possible to use the CPU usage as a universal indirect estimator of the power consumption, but more research is needed to find the right combination of features for this purpose.

4.2 Adaptive RAN Power Management in Serverless Environments

Effective energy optimisation in cloud-native and serverless Radio Access Networks (RANs) requires a detailed understanding of how individual system parameters influence total power consumption. Key determinants include the power consumed per radio port on the RRU, the configured MIMO level, the utilised bandwidth, the adopted TDD split ratio, slicing configuration, OSS user profile, user-generated traffic patterns, traffic duration, and the associated application behaviour. These factors collectively define the energy profile of the deployed RAN and are critical for the design of intelligent, adaptive power-management mechanisms. To assess these dependencies, the project employed the 5G/6G testbed infrastructure described in section 5.4.1. A comprehensive measurement campaign was carried out to quantify system-level behaviour, validate theoretical assumptions, and identify optimisation opportunities relevant for serverless and cloud-native deployments.

4.2.1 Energy Use Patterns on the 5G HW

The first phase of the evaluation investigated the influence of RAN component states and operational configurations on measured power consumption. Figure 4-24 provides an overview of the results, distinguishing between the RRH consumption (dark blue line) and the consumption attributable to the two power supplies of the IaaS environment (yellow and green lines).

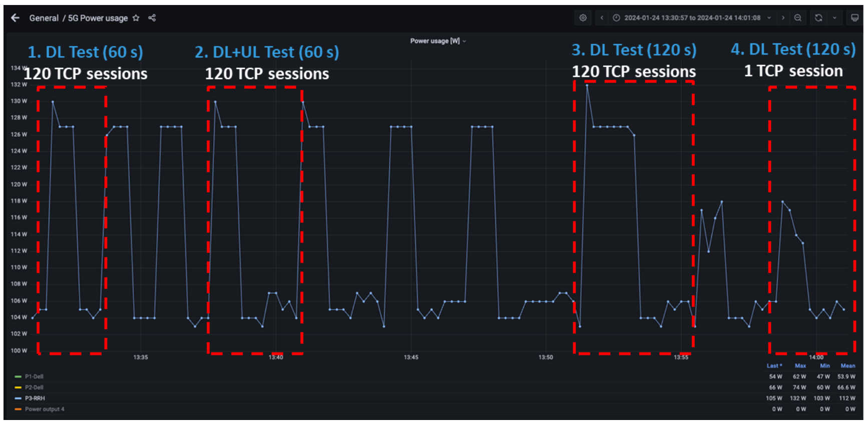


	Last *	Max
P1-Dell	48 W	61 W
P2-Dell	60 W	77 W
P3-RRH	69 W	139 W

Figure 4-24: Energy use patterns – 5G HW.

When the BBU was not active (Step 0), the RRU exhibited a baseline consumption of 68 W under the tested configuration (band n77, 100 MHz bandwidth, QAM 256 DL/UL, 27 dBm per port). Deployment of the 5G Core onto the IaaS platform (Step 1) affected only server-side consumption; RRU power remained unchanged. Activating the BBU (Step 2) with a UE attached in idle mode increased RRU consumption to 100 W under a 2×2 MIMO configuration, and subsequently to 105 W when the MIMO configuration was changed to 4×4 (Step 3).

Adjusting the TDD profile (Step 4 & 5) from a symmetric configuration to a DL-optimised configuration resulted in an increase to 108 W. During active user-traffic (Step 6) generation (60 seconds of TCP traffic followed by 60 seconds of idle time), consumption peaked at 130 W. The alternation between high (130 W) and idle (108 W) power states is clearly reflected in the measurement traces.



	Last *	Max	Min	Mean
P1-Dell	54 W	62 W	47 W	53.9 W
P2-Dell	66 W	74 W	60 W	66.6 W
P3-RRH	105 W	132 W	103 W	112 W
Power output 4	0 W	0 W	0 W	0 W

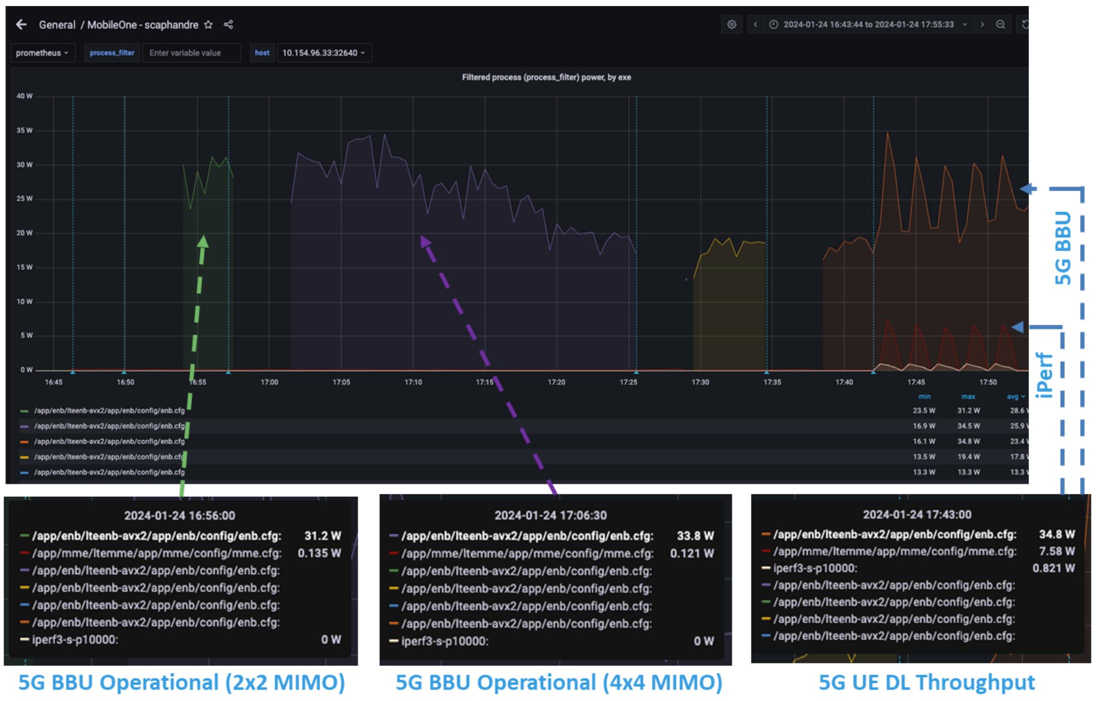
Figure 4-25: Dependences between user behaviour and application design on 5GS power usage.

Figure 4-25 further demonstrates the relationship between user behaviour, application-level design choices, and RRH energy usage. Short, intensive download phases—particularly with a high number of parallel TCP sessions—drive significantly higher consumption than idle periods. Upload traffic, in contrast, results in only a minor increase relative to idle consumption, illustrating that uplink processing is notably less energy demanding. Longer download durations proportionally extend the high-consumption plateau, while reducing the number of TCP sessions significantly lowers RRU load (less user load can be generated). These insights underscore the importance of application design and traffic pattern predictability in the context of energy-efficient mobile-network operation.

4.2.2 Energy Use Patterns on the 5G SW

To complement the hardware-level analysis, the Scaphandre measurement tool was used to evaluate power consumption at the software-component level, including virtualised BBU functions, the 5G Core, and traffic-generation applications (iPerf). The same test sequence used for hardware evaluation was applied to maintain methodological consistency.

As illustrated in Figure 4-26 software-component consumption scales directly with traffic intensity and the associated computational load. More complex MIMO configurations led to higher BBU consumption, while application-level tools such as iPerf exhibited consumption patterns that closely correlate with BBU workload. These results confirm the strong coupling between RAN functions and application traffic characteristics in cloud-native 5G/6G deployments.


General / MobileOne - scaphandre
2024-01-24 16:43:44 to 2024-01-24 17:55:33
prometheus
process_filter
Enter variable value
host
10.154.96.33:32640
Filtered process (process_filter) power, by exe
5G BBU
iPerf
5G BBU Operational (2x2 MIMO)
5G BBU Operational (4x4 MIMO)
5G UE DL Throughput

	min	max	avg
/app/enb/lteenb-avx2/app/enb/config/enb.cfg	23.5 W	31.2 W	28.6
/app/enb/lteenb-avx2/app/enb/config/enb.cfg	16.9 W	34.5 W	25.9
/app/enb/lteenb-avx2/app/enb/config/enb.cfg	16.1 W	34.8 W	23.4
/app/enb/lteenb-avx2/app/enb/config/enb.cfg	13.5 W	19.4 W	17.8
/app/enb/lteenb-avx2/app/enb/config/enb.cfg	13.3 W	13.3 W	13.3

```
2024-01-24 16:56:00
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:    31.2 W
/app/mme/ltemme/app/mme/config/mme.cfg:        0.135 W
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
iperf3-s-p10000:                                   0 W
```

```
2024-01-24 17:06:30
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:    33.8 W
/app/mme/ltemme/app/mme/config/mme.cfg:        0.121 W
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
iperf3-s-p10000:                                   0 W
```

```
2024-01-24 17:43:00
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:    34.8 W
/app/mme/ltemme/app/mme/config/mme.cfg:         7.58 W
iperf3-s-p10000:                               0.821 W
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
/app/enb/lteenb-avx2/app/enb/config/enb.cfg:
```

Figure 4-26: Energy Use Patterns – more complex MIMO configuration causes more power consumption (left, middle), 5G BBU component's power consumption correlates to application power consumption.

4.2.3 Advanced Experimentation

Building upon the initial energy-consumption characterisation, a structured set of optimisation mechanisms was experimentally validated. These mechanisms were utilised to manage RAN power usage in response to changing operational conditions, with relevance to the 6Green use case focusing on maintaining critical communication capabilities during energy-constraint scenarios.

The following mechanisms were evaluated: gradual cell shutdown, radio port output-power optimisation, cell bandwidth adaptation and MIMO-level adaptation. Each mechanism provides different saving potentials and implications for end-to-end network performance, which were analysed in detail.

Gradual Cell Shutdown

Gradual cell shutdown offers substantial energy savings by transitioning RRUs into standby mode (through CPRI link deactivation) or by completely powering off individual RRU units. This mechanism also reduces the corresponding BBU processing load for deactivated cells, thereby achieving significant system-wide energy reduction. It is particularly suited for low-traffic periods or scenarios where maintaining only minimal coverage is acceptable.

(B)5G Testbed Configuration:

- *Baseline: BBU with 2 × RRU (dual-cell).*
- *Cell configuration: n77, 3800 MHz, BW: 100 MHz, MIMO: 4×4, QAM256 DL/UL, TX Power: 26 dBm/port.*

Test procedure:

- *Cell shutdown via CPRI deactivation and RRU power-off.*

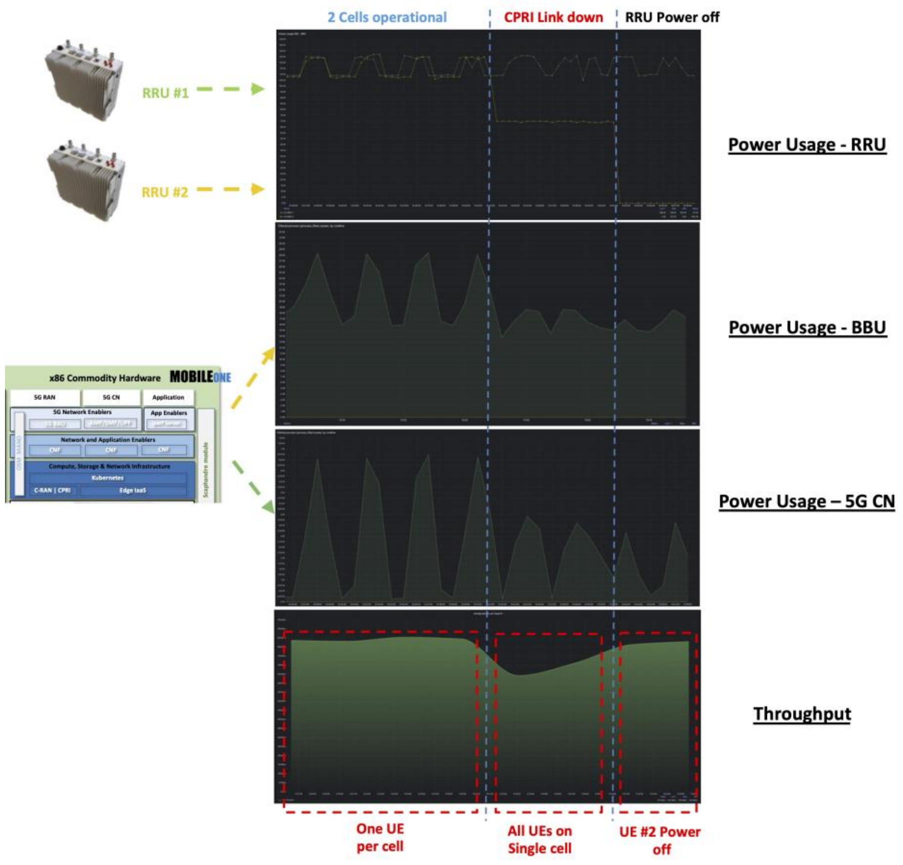


Figure 4-27: Energy Savings by Applying Gradual Cell Deactivation

In our test configuration (BBU with two RRUs operating in band n77 at 3800 MHz with 100 MHz bandwidth, 4×4 MIMO, and 26 dBm per port), deactivating one RRU reduced hardware consumption by up to 50%, decreased BBU processing by up to 33%, and reduced 5G Core processing by up to 42%. When two cells each served one UE, throughput reached approximately 850 Mbps per UE. When a single active cell served two UEs, throughput decreased to roughly 643 Mbps, representing an expected reduction given the operational constraints.

Radio Port Power Optimisation

Transmission-power optimisation revealed that maximum output power does not necessarily correspond to maximum throughput achieved by served UEs. Across the tested RRU port power levels (33, 31, 29, 27, 25, and 23 dBm), the highest throughput—950 Mbps—occurred at 27 dBm. This represents up to 121% improvement compared with the maximum-power configuration (33 dBm), which achieved only 430 Mbps due to increased signal distortion (reference UE was too close to the cell). Lowering the output power to 23 dBm produced 580 Mbps, offering reduced coverage but still acceptable performance.

(B)5G Testbed Configuration:

- *Baseline: BBU with 1 × RRU.*
- *Cell configuration: n77, 3800 MHz, BW: 100 MHz, MIMO: 4×4, QAM256 DL/UL.*

Test procedure:

- *Reduce TX power from 33, 31, 29, 27, 25 to 23 dBm,*

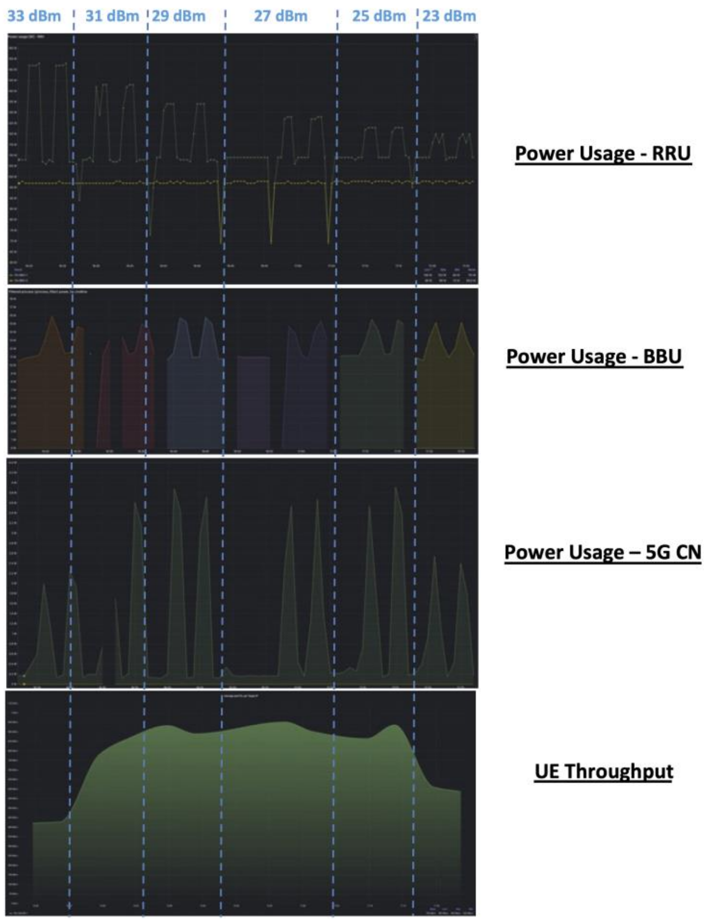


Figure 4-28: Radio Port Power Optimisation with Corresponding Throughput Gains

The optimisation resulted in up to 21% reduction in RRU hardware power consumption and up to 48% reduction in 5G Core processing due to lower achievable UE throughput. BBU consumption remained in the same range, as its base processing load is not influenced by the RRU transmit-power adjustments. This experiment illustrates a key finding: in the case when UEs are close to the cell tower, optimal operational efficiency is achieved through moderate rather than maximum transmission power.

Radio Bandwidth Optimisation

Bandwidth adaptation on the cell proved to be the most effective mechanism for reducing software-side energy consumption. Reducing bandwidth from 100 MHz to 20 MHz decreases the total number of resource blocks from 273 to 106 (a 61% reduction), resulting in proportionally lower BBU processing requirements.

(B)5G Testbed Configuration:

- *Baseline: BBU with 1 × RRU.*
- *Cell configuration: n77, 3800 MHz, MIMO: 4×4, QAM256 DL/UL, TX Power: 25 dBm/port.*

Test Procedure

- *Reduce cell Bandwidth from 100, 50 to 20 MHz.*

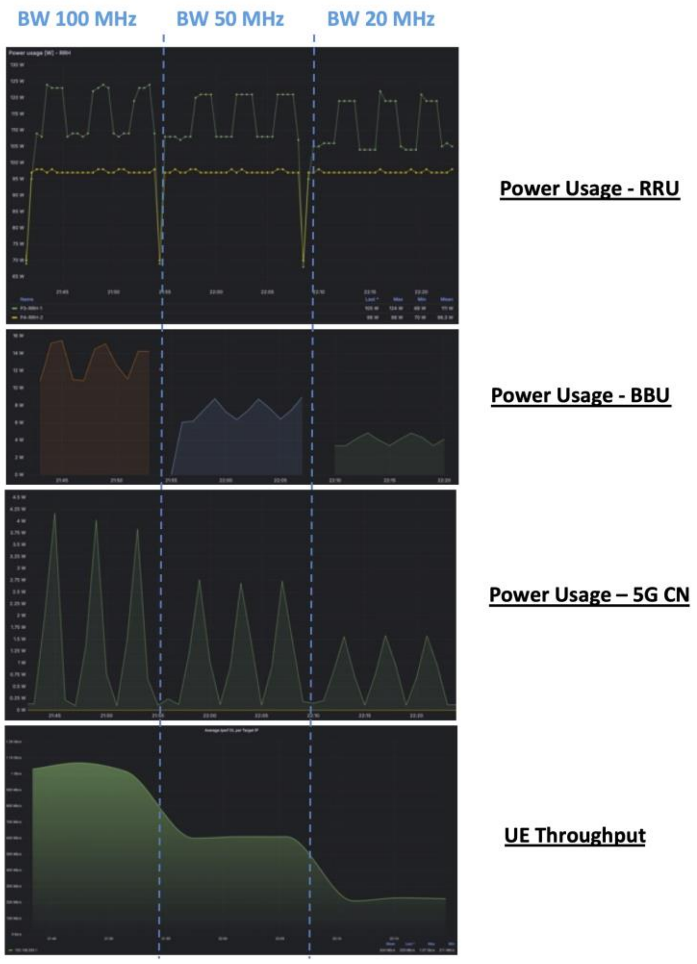


Figure 4-29: Energy Reduction Enabled by Bandwidth Downscaling

The tests showed a minor reduction in RRU hardware consumption (approximately 4%) but a substantial decrease in BBU and 5G Core consumption (up to 69% and 62% respectively). The throughput impact was proportional to the allocated bandwidth: 1 Gbps at 100 MHz, approximately 500 Mbps at 50 MHz, and 230 Mbps at 20 MHz. The 20 MHz configuration remains adequate for essential services such as emergency voice, messaging, and alert dissemination, making bandwidth optimisation particularly relevant for crisis-response scenarios.

MIMO Level Optimisation

Adjusting the MIMO configuration offers a balanced optimisation option that delivers both reduced energy consumption and improved radio link robustness in degraded radio environments. Lowering the MIMO level reduces the number of active RF chains at the RRU and significantly decreases spatial-processing requirements at the BBU.

(B)5G Testbed Configuration:

- *Baseline: BBU with 1 × RRU.*
- *Band: n77, 3800 MHz, BW: 100 MHz, QAM256 DL/UL, TX: 25 dBm/port.*

Test Procedure

- *Reduce MIMO from 4×4, 2×2 to SISO level.*

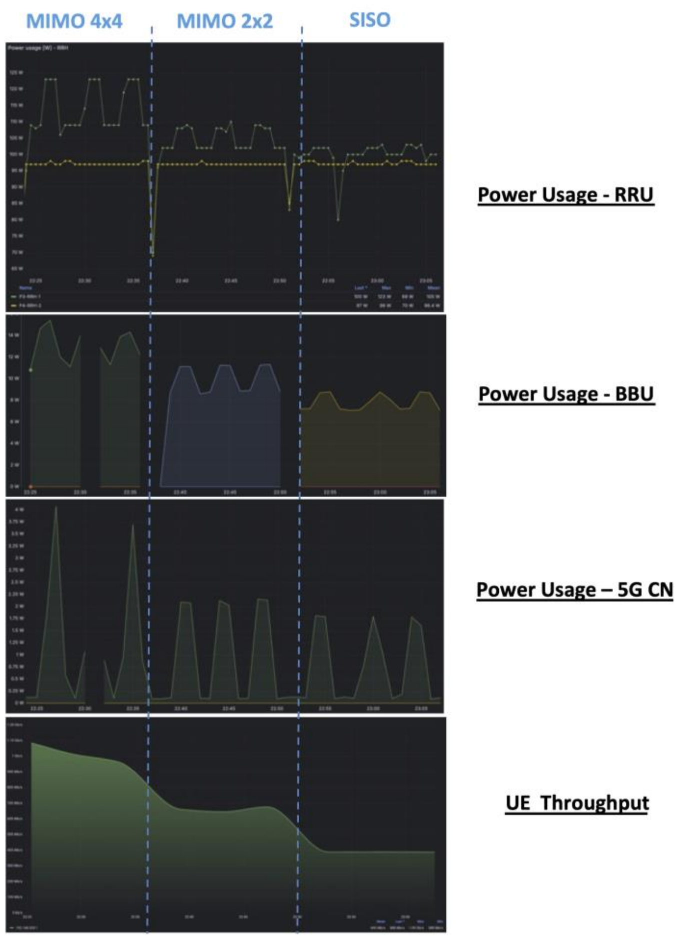


Figure 4-30: Power Efficiency Improvements via MIMO Downscaling

Measured savings included up to 16% reduction in RRU hardware consumption, up to 43% reduction in BBU consumption, and up to 55% reduction in 5G Core power consumption. Throughput decreased from approximately 1 Gbps (4×4 MIMO) to 670 Mbps (2×2 MIMO) and 390 Mbps (SISO). Importantly, the SISO configuration demonstrated the highest link stability, which is crucial under non-line-of-sight and infrastructure-degraded conditions.

4.2.4 Main findings

The validated mechanisms support context-aware optimisation strategies that can be selectively applied depending on the operational scenario. Gradual cell shutdown enables up to 50% energy savings on the RRU side (base station with two cells) with moderate QoE impact and is well suited for low-density base stations or emergency-only operation. Radio transmission-power optimisation delivers up to 25% savings with negligible—and under favourable conditions, such as UEs located near the cell site—even positive QoE impact. Radio-bandwidth reduction provides up to 70% savings and represents the most effective software-based mechanism, though it introduces significant capacity constraints. MIMO-level reduction yields up to 55% savings and enhances link robustness in challenging radio environments.

The following key conclusions emerged. First, the identified “throughput efficiency paradox” demonstrates that moderate transmit-power configurations can outperform high-power operation in both throughput and energy efficiency. Second, combining multiple mechanisms produces cumulative benefits, enabling up to 70%

total energy reduction in extreme energy-constrained scenarios. Third, the observed deterministic power-consumption patterns provide a strong foundation for automated AI/ML-based control.

The comprehensive measurement and validation activities conducted in this task confirm that the investigated mechanisms can jointly deliver up to 70% system-wide energy savings, making them suitable for highly energy-constrained contexts such as disaster-response operations. The work provides quantified performance impacts, identifies important cross-layer efficiency interactions, and establishes a reproducible methodological basis for AI/ML-enabled autonomous power management. As next steps, we will integrate these findings into higher-level management frameworks, validate the mechanisms under realistic crisis conditions as part of the use-case activities, and align them with renewable-energy systems and advanced battery-management solutions.

The complexity and multidimensional nature of RAN energy optimisation necessitate the adoption of AI-driven approaches. Prediction models can anticipate traffic behaviour and proactively adjust RAN configurations. Multi-objective reinforcement-learning methods can balance competing parameters such as coverage, QoE, and energy consumption. Context-recognition models enable the system to automatically identify operational states, while anomaly-detection models can reveal irregular consumption patterns or early signs of hardware degradation.

4.3 Energy-Aware Network Slice Management in O-RAN

This modular approach to service delivery achieved by network slicing is complemented by the disaggregation of the Radio Access Networks (RAN) architecture as e.g., suggested by the Open RAN (O-RAN) Alliance[17] in order to enable RAN openness and interoperability. This architecture divides the RAN into three key components: the Central Unit (CU), the Distributed Unit (DU), and the Radio Unit (RU), which can be deployed on open hardware and cloud nodes as VNFs. Network slicing in O-RAN, is intricately linked to the placement of RAN-specific Network Functions (NFs) in the RU, DU, and CU. By deploying DUs closer to RUs at the network edge, operators can reduce latency and improve the overall performance of RAN slices. However, in this regard, network slicing in O-RAN is mapped into a complex RU, DU, and CU resource allocation problem. Challenges arise in the dynamic allocation of these resources to support varying slice requirements and changing slice request patterns, while minimizing power consumption and reconfiguration costs associated with VNF migration towards improving slice admittance ratio [20]. Generally, VNF allocation is performed either proactively but assuming perfect forecasts of future slice admission requests for a quite long time horizon (e.g., [21]), or reactively upon arrival of the slice requests with future knowledge on traffic arrivals in an expected sense (e.g., [22], [23]). The above challenges underscore the need for innovative solutions to optimize resource utilization, minimize network delay and power consumption but also importantly enhance the robustness of O-RAN slicing deployments under uncertainties on future knowledge. This section presents our work that contributes towards this direction by solving the problem of optimal joint slice admission control and VNFs placement in the O-RAN modules with an iterative Model Predictive Control (MPC) strategy that allows considering updated forecasts of future slice arrivals. Also, it aims to shed light on the issue of minimizing the reconfiguration costs associated with optimizing multiple slice deployments, which are related to slice downtime (decreased slice availability), offering insights into strategies to streamline this process and ensure maximization of revenue during slice admission. We appropriately handle reconfiguration along the MPC iterations to improve slice admittance in an energy efficient way. Additionally, the proposed setting considers vendors' Quality of Service (QoS) issues such as end-to-end delays, but also, an overall green operation through accounting for power consumption costs.

[17] https://www.o-ran.org

4.3.1 System Architecture and Modeling

Figure 4-31 depicts the deployed O-RAN based architecture. In detail, micro-datacenters, namely Edge Clouds (ECs), are deployed at the network edge and serve as computing resources, in the proximity of the radio unit enabling low-latency processing and reducing backhaul traffic $\mathcal{E}$denote the set of ECs of the topology. Each EC, $e \in \mathcal{E}$, hosts a DU responsible for processing and managing network functions associated with specific network slices. The ECs are connected with the cell-cite, where an RU is deployed, via fronthaul (FH) connections, while midhaul (MH) links connect each EC with the Regional Cloud (RC) datacenter, denoted by $\mathcal{R}$, where the CU is deployed. The RC serves as a centralized computing resource for higher-level processing and coordination across multiple ECs. The FH links facilitate low-latency communication between the RU and the DUs, while MH links provide high-bandwidth connectivity between DUs and CU. For every $e \in \mathcal{E}$ the total computing capacity, in CPU cores is defined as $\mathrm{CE_e}$, while the corresponding parameter for the regional cloud is denoted by CR. Furthermore, transmission delay of the FH and the MH links is defined as $\delta_{r,e}$ and $\delta_{e,\mathcal{R}}$, $\forall e \in \mathcal{E}$, where r is the RU. Moreover, $\mathrm{CB_{F,e}}, \mathrm{CB_{M,e}}$ stand for the bandwidth of the FH and MH links associated with the EC, $e \in \mathcal{E}$, respectively.

4.3.2 Slice Request Model

In the proposed O-RAN-based system modeling, we consider a set *F* consisting of available VNFs, denoted by $v_f \in F$ that can be deployed to compose various network slices. It is important to note that certain VNFs, specifically the VNFs with IDs v_0, v_1, remain consistent across all network slice requests. In precise, VNFs v_0, v_1 are the initial VNFs used in every request *s*. When a network slice request *s* arrives, it is considered as an ordered set of elements of *F*, $F_s = \{v_0^s, v_1^s, \dots, v_f^s, \dots, v_{n_s}^s\} \subseteq F$, where $v_0^s = v_0 \in F$, $v_1^s = v_1 \in F$. Notably, each network slice request *s* is structured following the Service Function Chain (SFC) deployment model, where a specific execution sequence is defined [1]. This sequence dictates the order in which the VNFs are processed within the network slice.

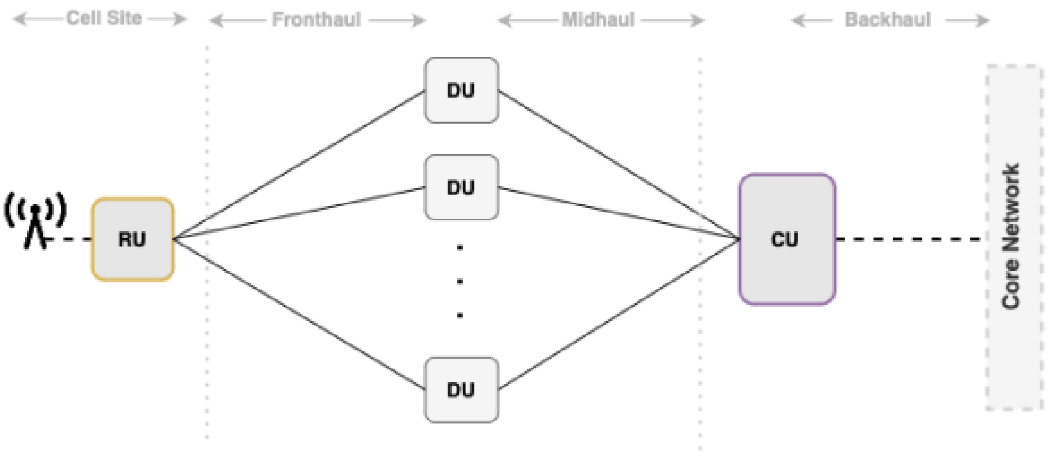


Figure 4-31: Proposed O-RAN-based Architecture.

Additionally, the compute and network related resource requirements are defined per slice *s*. For a VNF $v_f \in F_s$, there exist specific demands regarding CPU cores for the VNF deployment $c_{s,f}$ and the bandwidth for the link $(f-1, f)$ denoted as $b_{s,f}$. Furthermore, each request *s* arrives at a specific time t_s and has a holding time ht_s, indicating the duration for which the slice remains active once requested. Moreover, an end-to-end delay requirement $D_{max,s}$ and a priority value pr_s is defined for each slice request, reflecting its tolerance level for delay and importance, respectively. All notations used are summarized in Table 4. Let us also specify the following variables that play an important role in the problem formulation:

$$x^e_{s,f}(t) = \begin{cases} 1, & f \in F_s \text{ is placed on EC } e \in \mathcal{E} \text{ at t,} \\ 0, & \text{otherwise.} \end{cases}$$

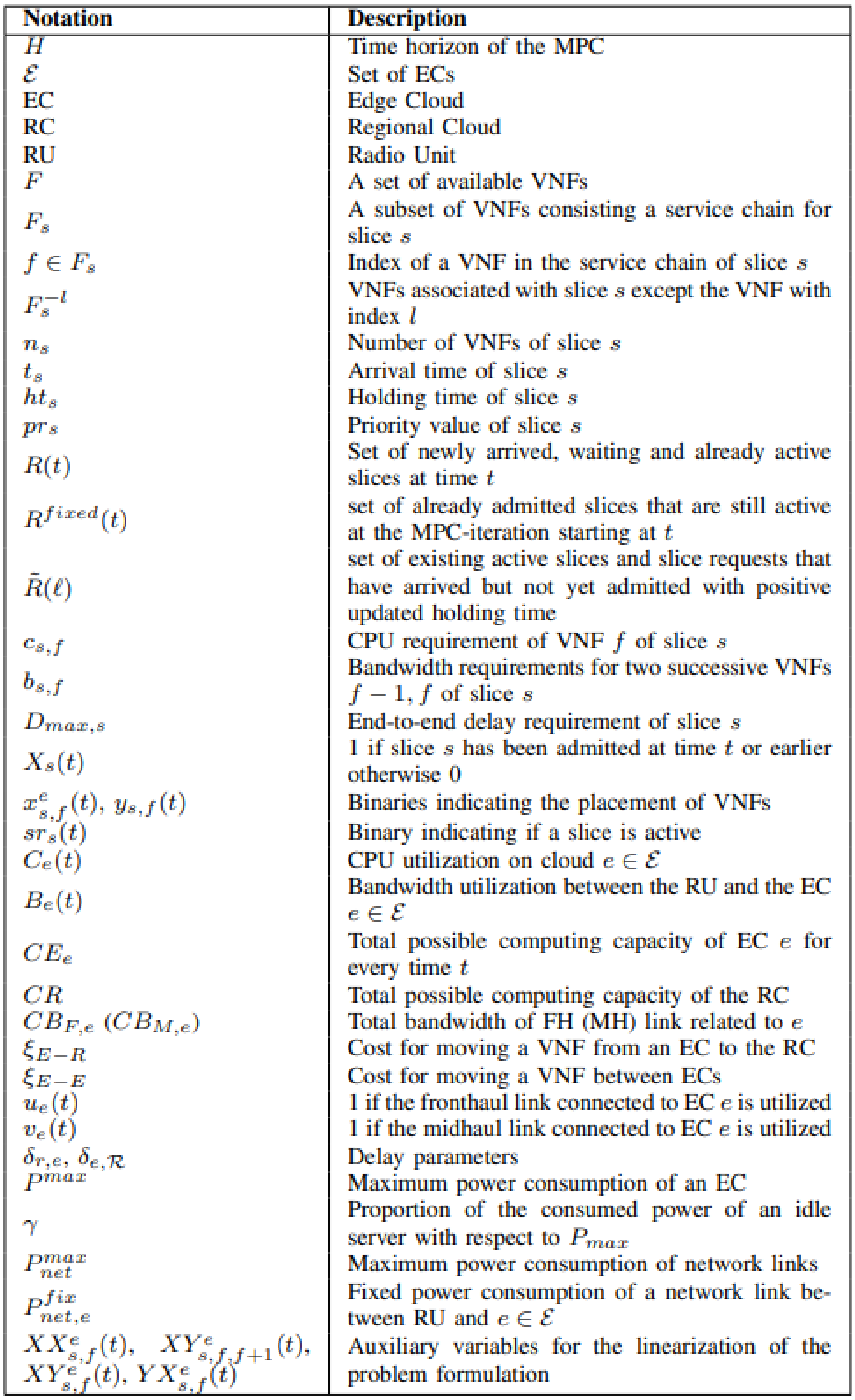
$$y_{s,f}(t) = \begin{cases} 1, & f \in F_s \text{ is placed on RC at t,} \\ 0, & \text{otherwise.} \end{cases}$$

x

$$sr_s(t) = \begin{cases} X_s(t), & t_s \leq t < (t_s + ht_s), \\ 0, & \text{otherwise.} \end{cases}$$

Furthermore, we consider that each EC hosts a single distributed computing unit. The VNF with index 0 is placed on the RU, and the remaining VNFs are placed either on an EC or the RC with the constraint that if a VNF is placed on an EC all its preceding VNFs in the path should be placed on ECs.

Table 4: Selective Notation and Description.



Notation	**Description**
H	Time horizon of the MPC
$\mathcal{E}$	Set of ECs
EC	Edge Cloud
RC	Regional Cloud
RU	Radio Unit
F	A set of available VNFs
F_s	A subset of VNFs consisting a service chain for slice s
$f \in F_s$	Index of a VNF in the service chain of slice s
F_s^{-l}	VNFs associated with slice s except the VNF with index l
n_s	Number of VNFs of slice s
t_s	Arrival time of slice s
ht_s	Holding time of slice s
pr_s	Priority value of slice s
$R(t)$	Set of newly arrived, waiting and already active slices at time t
$R^{fixed}(t)$	set of already admitted slices that are still active at the MPC-iteration starting at t
$\bar{R}(\ell)$	set of existing active slices and slice requests that have arrived but not yet admitted with positive updated holding time
$c_{s,f}$	CPU requirement of VNF f of slice s
$b_{s,f}$	Bandwidth requirements for two successive VNFs $f-1, f$ of slice s
$D_{max,s}$	End-to-end delay requirement of slice s
$X_s(t)$	1 if slice s has been admitted at time t or earlier otherwise 0
$x_{s,f}^e(t)$, $y_{s,f}(t)$	Binaries indicating the placement of VNFs
$sr_s(t)$	Binary indicating if a slice is active
$C_e(t)$	CPU utilization on cloud $e \in \mathcal{E}$
$B_e(t)$	Bandwidth utilization between the RU and the EC $e \in \mathcal{E}$
CE_e	Total possible computing capacity of EC e for every time t
CR	Total possible computing capacity of the RC
$CB_{F,e}$ $(CB_{M,e})$	Total bandwidth of FH (MH) link related to e
ξ_{E-R}	Cost for moving a VNF from an EC to the RC
ξ_{E-E}	Cost for moving a VNF between ECs
$u_e(t)$	1 if the fronthaul link connected to EC e is utilized
$v_e(t)$	1 if the midhaul link connected to EC e is utilized
$\delta_{r,e}$, $\delta_{e,\mathcal{R}}$	Delay parameters
P^{max}	Maximum power consumption of an EC
γ	Proportion of the consumed power of an idle server with respect to P_{max}
P_{net}^{max}	Maximum power consumption of network links
$P_{net,e}^{fix}$	Fixed power consumption of a network link between RU and $e \in \mathcal{E}$
$XX_{s,f}^e(t)$, $XY_{s,f,f+1}^e(t)$, $XY_{s,f}^e(t)$, $YX_{s,f}^e(t)$	Auxiliary variables for the linearization of the problem formulation

4.3.3 Problem Formulation

System Dynamics and Constraints

The CPU utilization of each EC evolves as follows:

$$C_e(t+\Delta\tau) = C_e(t) + \sum_{s\in R(t+\Delta\tau)} \sum_{f\in F_s} (x^e_{s,f}(t+\Delta\tau) - x^e_{s,f}(t))c_{s,f}.$$

as obtained after the calculations provided analytically in [24]. In similar lines, we compute the evolution equation of the bandwidth utilization of the FH links as:

$$B_e(t+\Delta\tau) = B_e(t) + \sum_{s\in R(t+\Delta\tau)} (x^e_{s,1}(t+\Delta\tau) - x^e_{s,1}(t))b_{s,1}.$$

Next, we shorty describe and provide the remaining system constraints:

First, the aggregate of the computing resources to bind in any EC or the RC has to be lower than the total possible computing capacity of the corresponding cloud, which is expressed as follows:

$$\sum_{s\in R(t)} \sum_{f\in F_s} x^e_{s,f}(t)c_{s,f} \leq CE_e, \ \forall e \in \mathcal{E},$$

$$\sum_{s\in R(t)} \sum_{f\in F_s} y_{s,f}(t)c_{s,f} \leq CR.$$

A slice uses a link between the RU and an EC, if its second VNF (i.e., with index 1) is placed in this EC. The bandwidth constraints for the FH and MH links are expressed for every time *t* as follows.

$$\sum_{s\in R(t)} x^e_{s,1}(t)b_{s,1} \leq CB_{F,e}, \ \forall e \in \mathcal{E},$$

$$\sum_{s\in R(t)} \sum_{f\in F_s^{-n_s}} x^e_{s,f}(t)y_{s,f+1}(t)b_{s,f+1} \leq CB_{M,e}, \ \forall e \in \mathcal{E}.$$

For every admitted slice, its first VNF (i.e., with index 0) is placed in the RU:

$$\sum_{\forall e\in\mathcal{E}} x^e_{s,0}(t) = 0, \ \forall s \in R(t),$$

$$y_{s,0}(t) = 0, \ \forall s \in R(t).$$

In addition, a VNF *f*, of an admitted slice *s*, can be allocated either to a single EC or the RC at every time i.e.,

$$\sum_{e\in\mathcal{E}} x^e_{s,f}(t) + y_{s,f}(t) = sr_s(t), \ \forall s \in R(t), \forall f \in F_s^{-0}.$$

Moreover, for an admitted slice, the VNF 1 should be placed in an EC, which is guaranteed if it cannot be placed in the RC, i.e.,

$$y_{s,1}(t) = 0, \ \forall s \in R(t).$$

Under the assumptions of service chaining and colocation, if for an admitted slice *s*, a VNF *f* is placed in an EC, the VNFs preceding *f* in the service chain should be also placed in the same EC. Similarly, if a VNF is placed in the RC, its successive VNFs in the service chain of the slice should be also placed in the RC.

Therefore,

$$x^e_{s,f}(t) \leq x^e_{s,f-1}(t),\ \forall f \in F_s^{-0,1}, \forall s \in R(t), \forall e \in \mathcal{E},$$
$$y_{s,f}(t) \leq y_{s,f+1}(t),\ \forall f \in F_s^{-n_s}, \forall s \in R(t).$$

The total delay imposed by FH and MH links at any time is bounded as follows:

$$\sum_{e\in\mathcal{E}} x^e_{s,1}(t)\delta_{r,e} + \sum_{f\in F_s^{-n_s}} \sum_{e\in\mathcal{E}} x^e_{s,f}(t) y_{s,f+1}(t)\delta_{e,R} \leq D_{\max,s},\ \ \forall s \in R(t).$$

A FH link is considered utilized only if one or more slices have placed their VNFs with index 1 in its corresponding EC, i.e.:

$$u_e(t) \geq x^e_{s,1}(t),\ \ \forall s \in R(t), \forall e \in \mathcal{E},$$
$$u_e(t) \in \{0,1\},\ \ \forall e \in \mathcal{E}.$$

A MH link is considered utilized if for any pair of two successive VNFs of any slice, one is placed in the EC and the other on the RC, i.e.,

$$v_e(t) \geq x^e_{s,f}(t) y_{s,f+1}(t),\ \ \forall s \in R(t), \forall f \in F_s^{-n_s}, \forall e \in \mathcal{E},$$
$$v_e(t) \in \{0,1\},\ \ \forall e \in \mathcal{E}.$$

Finally, a slice that gets admitted at time should be considered admitted for its entire control lifecycle, i.e.,

$$X_s(t+\Delta\tau) \geq X_s(t),\ \ \forall s \in R(t).$$

Objective Function

To define the objective function we consider three factors, namely: (i) the revenue obtained from slice acceptance, (ii) the cost deriving from reallocating already accepted slices, and (iii) the power consumption of the ECs and the network links that are utilized for the slice deployment. The revenue of a slice acceptance at time *t* is

$$ReV(t) = \sum_{s\in R(t)} sr_s(t) \cdot pr_s.$$

For the reallocation cost both VNFs moving from an EC to the RC or vice versa and those VNFs that move from an EC to another are considered. The instantaneous reallocation cost of VNFs from an EC to the RC or vice versa is expressed as:

$$ReC^{E-R}(t) = \sum_{s\in R(t), f\in F_s, e\in\mathcal{E}} (x^e_{s,f}(t) y_{s,f}(t+\Delta\tau) + y_{s,f}(t) x^e_{s,f}(t+\Delta\tau))\xi_{E-R},$$

while the reallocation cost from an EC to a different EC is expressed by

$$ReC^{E-E}(t) = \sum_{s\in R(t), f\in F_s, e\in\mathcal{E}, i\in\mathcal{E}^{-e}} x^e_{s,f}(t) x^i_{s,f}(t+\Delta\tau)\xi_{E-E}.$$

Regarding the power consumption cost, we follow the modelling of a power efficient VNF placement approach from the literature [25]. In case of ECs, it is given by:

$$PC^{EC}(t) = \sum_{e\in\mathcal{E}} \left(u_e(t)\gamma P^{max} + (1-\gamma)\frac{C_e(t)}{CE_e} P^{max} \right).$$

In case of links that connect the RU with an EC, it is formulated as:

$$PC^{RU-E}(t) = \sum_{e\in\mathcal{E}} \left(u_e(t) P^{fix}_{net,e} + \frac{B_e(t)}{CB_{F,e}} P^{max}_{net} \right).$$

Finally, in case of links that connect an EC with the RC, it can be written as:

$$PC^{E-R}(t) = \sum_{e\in\mathcal{E}} v_e(t) P^{fix}_{net,e}.$$

Optimization Problem – Problem 1:

$$\max \sum_{t=\ell:\Delta\tau:\ell+(H-1)\Delta\tau} \Big(ReV(t) - ReC^{E-R}(t) - ReC^{E-E}(t) - PC^{EC}(t) - PC^{RU-E}(t) - PC^{E-R}(t) \Big) \cdot \Delta\tau$$

subject to:

all system dynamics and constraints expressed above

and

$$X_s(t) \in \{0,1\},\ x^e_{s,f}(t), y_{s,f}(t) \in \{0,1\},$$
$$\forall s \in R(t), \forall f \in F_s, \forall e \in \mathcal{E}, \forall t \in \{\ell, \ell+\Delta\tau.., \ell+(H-1)\Delta\tau\},$$

where $C_e(l), B_e(l), \forall e \in \mathcal{E}$ are given. The optimization problem is mixed integer quadratically constrained problem with quadratic objective. Next, we apply Watters' linearization [26] on the quadratic terms in both the objective function and the constraints and the problem takes a MILP form.

4.3.4 Proposed Solution via Model Predictive Control (MPC)

To perform dynamic optimal slice admission and resource allocation on admitted slices, we solve the Problem 1 in a Model Predictive Control (MPC) fashion as illustrated in Figure 4-32. The control period starts at t_0 where no slices have arrived yet and thus no computing and bandwidth resources have been yet allocated. Problem 1 is then solved with initial time t_0 and a horizon of *H* time intervals in the future each of duration $\Delta\tau$. The number of slices and their arrival times within the future time horizon *H* is unknown and forecasts are used. In this work, forecasts are considered given by an external forecasting tool. The decisions about slice admittance and resource allocation are obtained for all time intervals within the horizon *H*. However, we apply only the decisions for time t_0 and disregard all other decisions for future times. By the time we apply the decisions we also observe which slices actually arrived. For slices that were forecasted to arrive but did not, we cancel any related resource allocation decision. For slices that arrived without being expected, we also do not allocate resources as otherwise infeasibilities and high costs may emerge. At the next decision time, i.e., $t_0 + \Delta$, the process is repeated. In particular, we observe the updated states regarding the computing resources of the ECs and the RC as well as the bandwidth of the links. Also, updated forecasts of the number and arrival times of new slices are obtained for a time horizon equal again to *H* time intervals in the future each of duration $\Delta\tau$. However, slices that have been already accepted at time t_0 or earlier, should continue providing service at time $l = t_0 + \Delta$, if their updated holding time is positive. This requirement cannot be directly handled by Problem 1 and necessitates the additional constraint $X_s(\ell) = 1, \forall s \in R^{Fixed}(\ell)$, with $R^{Fixed}(l)$ the set including all slices satisfying $s \in \widetilde{R}(l-\Delta\tau)$ and $ht_s \geq \Delta\tau$ and $X_s(l-\Delta\tau) = 1$. In addition, the Problem 1 should be adapted in order to account for the potential re-allocation costs of slices between times two consecutive decision times. To do so we introduce new binary parameters $x^{e,Fixed}_{s,f}, y^{Fixed}_{s,f}$, for all slices $s \in R^{Fixed}(l)$ with values set as $x^{e,Fixed}_{s,f} = x^e_{s,f}(l-\Delta\tau)$, $y^{Fixed}_{s,f} = y_{s,f}(l-\Delta\tau)$. Based on the above, we formulate Problem 2 that is an adapted version of Problem 1 for being integrated in an MPC framework. Algorithm 1 presents a pseudo-code of the solution process.

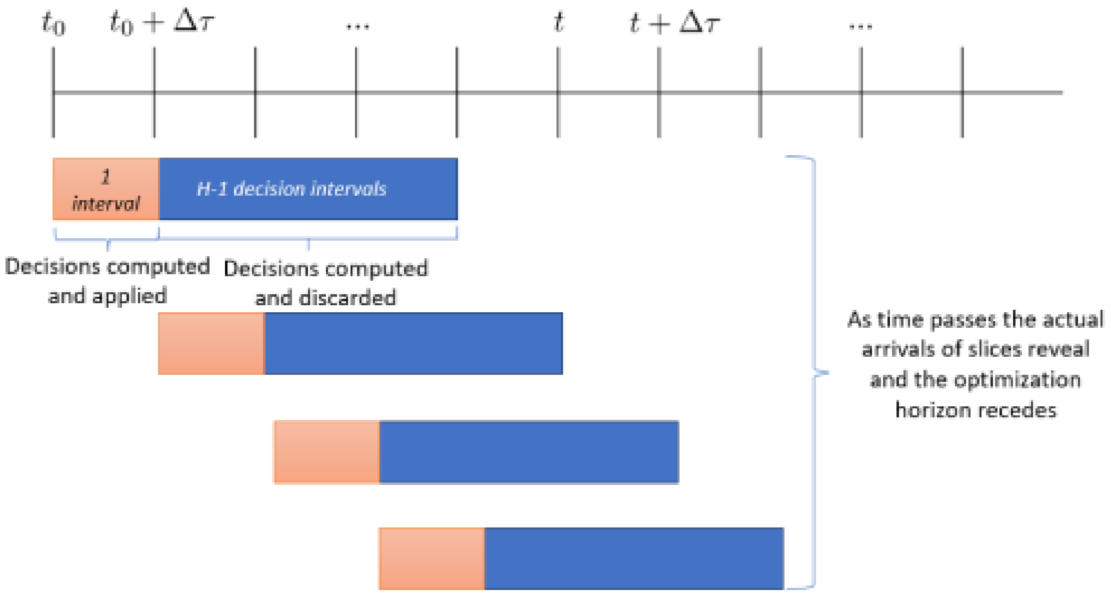

t_0
$t_0+\Delta\tau$
...
t
$t+\Delta\tau$
...
1 interval
H-1 decision intervals
Decisions computed and applied
Decisions computed and discarded
As time passes the actual arrivals of slices reveal and the optimization horizon recedes

Figure 4-32: MPC Iterations.

Optimization Problem to be integrated in an MPC Framework – Problem 2:

$$\max \sum_{t=\ell:\Delta\tau:\ell+(H-1)\Delta\tau} \Big(ReV(t) - ReC^{E-R}(t) - ReC^{E-E}(t) - PC^{EC}(t) - PC^{RU-E}(t) - PC^{E-R}(t) \Big) \cdot \Delta\tau - REC^{E-R,Init}(\ell)\cdot \mathbf{1}_{\ell>t_0} - REC^{E-E,Init}(\ell)\cdot \mathbf{1}_{\ell>t_0}$$

subject to:

all network dynamics and constraints above

$$X_s(t)\in\{0,1\},\ x^{e}_{s,f}(t), y_{s,f}(t)\in\{0,1\},$$
$$\forall s\in R(t), \forall f\in F_s, \forall e\in\mathcal{E}, \forall t\in\{\ell,\ell+\Delta\tau..,\ell+(H-1)\Delta\tau\},$$

$$\text{if } \ell>t_0 \text{ include } X_s(\ell)=1, \forall s\in R^{Fixed}(\ell),$$

where:

$$\mathbf{1}_{\ell>t_0}=\begin{cases}1, \text{if } \ \ell>t_0\\ 0, \ \ \text{otherwise,}\end{cases}$$

$$ReC^{E-R,Init}(\ell)=\sum_{s\in R^{Fixed}(\ell),f,e}\left(x^{e,Fixed}_{s,f}y_{s,f}(\ell)+y^{Fixed}_{s,f}x^{e}_{s,f}(\ell)\right)\xi_{E-R}\cdot\Delta\tau,$$

$$ReC^{E-E,Init}(\ell)=\sum_{s\in R^{Fixed}(\ell),f}\sum_{e\in\mathcal{E}}\sum_{i\in\mathcal{E}-e}x^{e,Fixed}_{s,f}x^{i}_{s,f}(\ell)\cdot\Delta\tau.$$

Algorithm 1 Model Predictive Control for Dynamic Resource Allocation on Network Slices

```
Input: Time horizon $H$; Initial time $t_0$; All parameters related to the $RU$, $EC$, $RC$, including power consumption, bandwidth, computing availability, re-allocation cost parameters;
  procedure MPC
  Initializations:
    $\ell \leftarrow t_0$; $C_e(t_0) \leftarrow 0$; $B_e(t_0) \leftarrow 0, \forall e \in \mathcal{E}$;
    while not the end of control period do
        1. Observe the state variables, $C_e(\ell)$, $B_e(\ell), \forall e \in \mathcal{E}$.
        2. If $\ell > t_0$, observe already active slices with positive updated holding time, $ht_s \leftarrow ht_s - \Delta\tau$, forming the set $R^{Fixed}(\ell)$. Also, compute the binary parameters $x_{s,f}^{e,Fixed}, y_{s,f}^{Fixed}$, for all slices $s \in R^{Fixed}(\ell)$.
        3. Receive updated forecasts of arriving slices for a horizon $H$, i.e., for times $\{\ell, \ell+\Delta\tau, ..., \ell+(H-1)\Delta\tau\}$;
        4. Solve Problem 2 and obtain the main optimization variables, i.e., $X_s(\ell)$, $x_{s,f}^e(\ell)$, $y_{s,f}(\ell)$ for all $s \in R(\ell)$, all related VNFs $f$ and all times $\{\ell, \ell+\Delta\tau, ..., +(H-1)\Delta\tau\}$;
        5. Observe the realizations of the uncertain quantities at the current decision interval $\ell$, i.e., $\tilde{R}(\ell)$;
        6. Keep the decisions only for time $\ell$, i.e., $X_s(\ell)$, $x_{s,f}^e(\ell)$, $y_{s,f}(\ell)$ for all $s \in \tilde{R}(\ell)$ (and related VNFs) and discard those of all future time slots, i.e., $\{\ell+\Delta\tau, ..., \ell+(H-1)\Delta\tau\}$. If a slice $s$ was forecasted to arrive but in reality did not or the vice versa set $X_s(\ell) = x_{s,f}^e(\ell) = y_{s,f}(\ell) = 0, \forall e \in \mathcal{E}, f \in F_s$.
        7. Update the state variables $C_e(\ell)$, $B_e(\ell)$ for the next decision interval $(\ell+\Delta\tau)$ using the equations of Section IV-B for slices in $\tilde{R}(\ell)$.
        8. $\ell \leftarrow \ell + \Delta\tau$
    end while
  end procedure
```

4.3.5 Evaluation Results (MPC)

<u>Evaluation Setting:</u>

This section presents the assessment of the proposed MPC-based solution approach for the problem 1. For the implementation of the simulation environment, version 3.10 of Python programming language is used. We follow an object-oriented programming approach, defining one class for the slice request model and one class for implementing the solution methods, i.e., the proposed mpc-based method and the alternative solutions. The substrate network parameters are involved in the solution class. To solve the optimization problem we use Gurobi solver, specifically, the gurobipy Python package. The parameter values are given in Table 5 and 4. The substrate network consists of three ECs and one RC. We consider two types of slices, URLLC and eMBB. VNF requirements adhere to a typical paradigm commonly for cloud service providers. These requirements manifest in three distinct flavors, denoted as small, medium, and large. Each flavor corresponds to varying levels of resource demands, particularly in terms of CPU cores for our modeling. Specifically, the CPU demand per flavor is specified as 2 cores for small, 4 for medium, and 8 for large. In the context of the simulation process, a flavor is chosen equiprobably for each VNF of every slice. We consider that the number of slice requests is equal to 15 over a time horizon of 12 time units. In order to perform a fair evaluation between the distinct approaches, we assume that the holding time of every deployed slice could not exceed the 12 timesteps setup which reflect to 24 hours of deployment time.

Table 5: Network parameters.

Parameter	**Value**
Number of ECs	3
EC, RC capacity	16, 64 cores
FH link capacity, delay	2Gbps, 4ms
MH link capacity, delay	4Gbps, 8ms
γ	0.8
P^{max}	2000W
P^{max}_{net}	200 W
P^{fix}_{net}	160 W

Table 6: Slice parameters.

Attributes	**Slice type**	
	URLLC	**eMBB**
$D_{max,s}$	25 ms	50 ms
$c_{s,f}$	$\in \{2,4,8\}$	$\in \{2,4,8\}$
$b_{s,f}$	100 Mbps	200 Mbps
Request arrival times t_s	$\mathcal{U}\{1,12\}$	$\mathcal{U}\{1,12\}$
Holding time ht_s	$\min(\mathcal{U}\{3,6\},\zeta_s)$	$\min(\mathcal{U}\{3,6\},\zeta_s)$
Number of requests	55% of total	45% of total
Normalized priority	3 per time-slot	2.4 per time-slot

We generate forecasts for the time arrival of requests using the following forecasting method. Initially, the arrival time of requests is sampled from a discrete uniform distribution over the optimization horizon, H. At each time slot of the control period, we solve the Problem 2 and obtain the decision variables. We consider that our forecasting method generates forecasts that are inaccurate with probability 10%. In this context, we define two forecasting scenarios, namely, (i) Less accurate forecasting scenario: In this scenario, the arrival time of not yet realized slice requests is resampled from a discrete uniform distribution over the horizon. (ii)Highly accurate forecasting scenario: Under the highly accurate forecasting scenario, *20%* of the expected requests to arrive resample their time arrival. All the simulations are executed in an Ubuntu 20.04 virtual machine with 8 vcpus and 8GB of RAM of an Intel(R) Xeon(R) CPU@2.10GHz server.

Evaluation Metrics:

The evaluation focuses on comparing the performance of three distinct methods: the proposed MPC solution, an MPC variant that avoids VNF reallocation (MPC-NR), and a one-shot optimization approach that decides the admission of slice requests at the first time slot for the entire horizon (One Shot). These methods were tested under the two different settings of slice request forecasting that were discussed above, in order to assess their robustness and adaptability to dynamic changing of slice request demands. The evaluation metrics for the performance assessment are:

Acceptance Ratio: The acceptance ratio measures the percentage of admitted slices by a certain time step, determined by the active slice subset, which includes ongoing requests yet to expire. It reflects the system's efficacy in handling incoming slice demands amidst existing deployment commitments.

Objective Value: This metric represents the optimization objective value achieved by each method based on the actual realization of slice requests, offering insights into their efficiency in resource allocation and utilization during the slice requests admission.

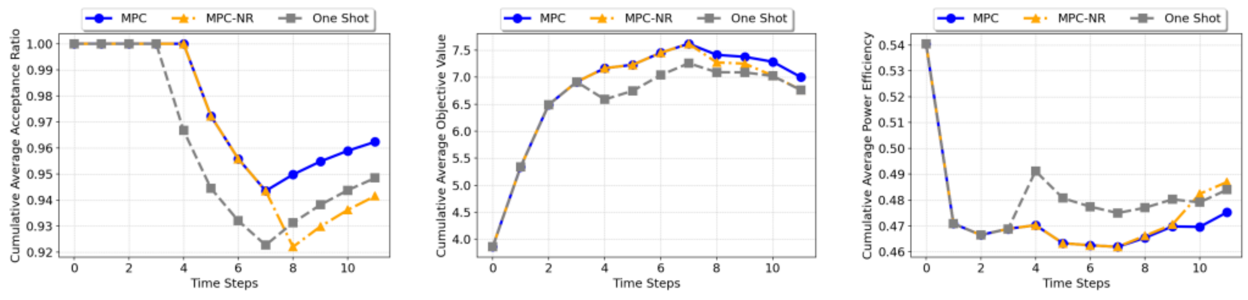
Power Efficiency: This is defined as the ratio of revenue generated by the admitted slices over the total power consumption of the compute and network counterparts of the substrate network. The inverse of power efficiency signifies the system's effectiveness in conserving energy, with lower values denoting higher power efficiency.

<u>Discussion on the Results:</u>

Figure 4-33 and Figure 4-34 present the cumulative average of the above evaluation metrics computed over a 12-time step horizon for the two cases of the forecasting scenarios, aiming to provide a comprehensive overview of the performance trends observed across the simulation. In scenarios with favorable forecast conditions, marginal differences are observed between the solution methods. However, upon closer examination, MPC demonstrates its adaptability over the prediction horizon, particularly in achieving higher acceptance ratios, as shown in Figure 4-33a. At the same time, it maintains optimal values for other key metrics compared to MPC-NR and One Shot solutions (Figure 4-33b,c), showcasing its ability to adjust resource allocation decisions regarding VNF placement, while achieving to maintain low consumption power of the compute and network counterparts.

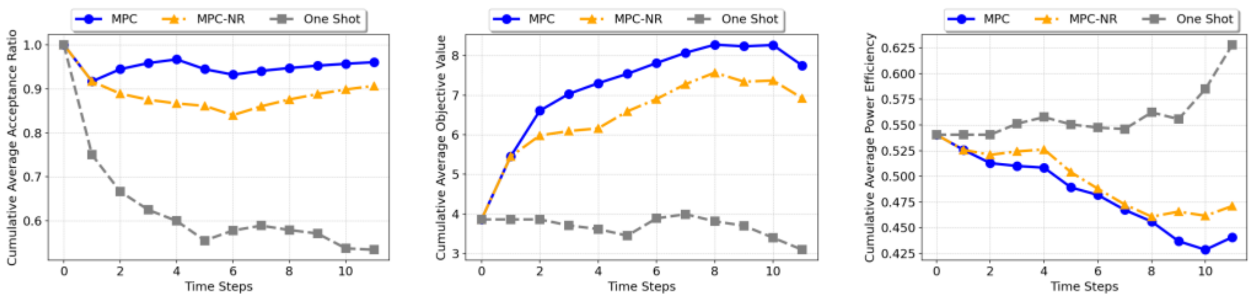
The efficacy of the MPC approach becomes more evident in less accurate forecast scenarios. The evaluation results regarding this scenario are shown in Figure 4-34. In more detail, despite inherent uncertainties, MPC consistently outperforms One Shot optimization method, highlighting its robustness and resilience to forecast inaccuracies. Moreover, compared to the MPC solution that totally eliminates the reallocation of VNFs, namely the MPC-NR, the proposed MPC approach maintains a substantial performance advantage across all evaluated metrics. More precisely, the optimal resource utilization is highlighted in Figure 4-34a, where the cumulative average of acceptance ratio is much higher than the other approaches from very early during the evaluation period and maintained for the whole horizon, as reflected in Figure 4-34b. It is worth mentioning, that despite the higher acceptance ratio, which entails to increased resource demand, the proposed MPC approach still outperforms the MPC-NR and One Shot methods in terms of power efficiency (Figure 4-34c).

The observed performance disparities underscore the significance of proactive and adaptive resource allocation strategies in dynamic network environments. While traditional optimization methods may suffice under ideal conditions, the inherent uncertainty of real-world scenarios necessitates more sophisticated approaches. The MPC ability to leverage forecast information to anticipate demand fluctuations and proactively optimize resource allocation decisions is a key determinant of its efficacy on slice admission in O-RAN-based architectures. Furthermore, the performance advantage of the MPC-based approach over MPC-NR reveals the importance of considering reallocation in dynamic resource allocation strategies. By factoring in these costs, the MPC framework effectively manages the trade-offs between resource usage optimization and the operational overhead associated with reallocating and migrating VNFs. This ensures optimal resource utilization while ensuring higher slice availability with minimal management complexities from the infrastructure provider's perspective.



(a) Cumulative average acceptance ratio. (b) Cumulative average objective value. (c) Cumulative average power efficiency.

Figure 4-33: Comparative evaluation results under highly accurate forecasting scenarios.



(a) Cumulative average acceptance ratio. (b) Cumulative average objective value. (c) Cumulative average power efficiency.

Figure 4-34: Comparative evaluation results under less accurate forecasting scenarios.

In future work, we are going to try adopting the abovementioned formulation in core network and serverless scenarios. Especially, for serverless computing paradigm, we will try to develop a two-level virtualization mechanism that provides a virtual network to an application (referred to as the related slice) and manages the serverless application deployment within the virtual network. By integrating the abovementioned forecasting mechanism with a workload estimator related to the application, we can work on creating a scaling approach for VNF replicas and virtual network resources.

4.3.6 Proposed Solution using Reinforcement Learning (RL)

In this section, we consider a simpler version of the optimization problem for joint admission control and resource allocation of network slices in the proposed O-RAN architecture, where the reconfiguration of the already admitted slices is being deactivated, by forbidding the reallocation of deployed VNFs.

A Markov Decision Process (MDP) is a typical framework to describe decision-making problems in a stochastic environment. An MDP consists of the set of states ,S, the set of actions A, a state transition function P, which indicates the probability $P(s' \mid s, a)$ of obtaining the state s' when taking action a from the state s, the reward function $r : S \times A \rightarrow$ R and the discount factor γ. Specifically, the policy $\pi : S \times A \rightarrow [0,1]$ indicates the probability of choosing the action $a \in A$ from state $s \in S$. The agent's objective is to learn an optimal policy $\pi^{\cdot}$, which maximizes the expected return $E\left[\sum_{\{k=0}^{\infty} \gamma^k r_k\right]$, where r_k is the reward that the agent receives after the k^{th} RL decision step.

In our setting, slice requests arrive at the agent and the agent decides whether to accept each one of them in order of their arrival times. Slice requests may arrive at the same time slot at the agent where in this case ties break arbitrarily. To avoid confusion, the indicator k denotes the RL-based decision step for accepting or not a slice corresponding to a request, whereas t_k denotes the time slot of the control window where the

RL decision k takes place. Multiple RL decision steps $k,\ k+1,\ \dots,\ m$ may refer to the same time slot $t_k = t_{k+1} = \dots = t_m$ in the control window when concurrent slice request arrivals take place. For two RL decision steps k and m with $k < m$ it should hold that $t_k \leq t_m$. Furthermore, slices that were not accepted are given as input to the agent at the next time slot if their holding time has not expired. In this case, they are considered as new slice requests with properly decreased holding times.

State: The state of the agent at the RL-decision step k is the tuple $(AC_k,\ AB_k,\ AT_k,\ SI_k,\ t_k)$, where $AC_k = \left[AC_1(k),\ \dots,\ AC_{|\mathcal{E}|}(k),\ A_{RC}(k)\right]$, collecting the available capacity of all ECs and of the RC; $AB_k = \left[AB_{FH,1}(k),\ \dots,\ AB_{FH,|\mathcal{E}|}(k),\ AB_{MH,1}(k),\ \dots,\ AB_{MH,|\mathcal{E}|}(k)\right]$ denotes the available bandwidth capacity of FH and MH links; $AT_k = \left[AT_{EC,1}(k),\ \dots,\ AT_{EC,|\mathcal{E}|}(k),\ AT_{MH,1}(k),\ \dots,\ AT_{MH,|\mathcal{E}|}(k)\right]$ collects the remaining times that each EC or each MH link will remain active according to the configuration at step k, t_k is the current time slot of the control horizon and SI_k contains necessary information regarding the slice about to be processed. In particular, $SI_k = \left(pr_s,\ D_{max,s},\ ht_s,\ c_{s,1},\ \dots, c_{s,n_s},\ b_s,\ \dots,\ b_{s,n_s}\right)$ assuming that slice s is examined at step k.

Action: The agent jointly decides which EC will serve the input slice as well as the number of VNFs of the slice placed at the chosen EC. Specifically, the agent has to decide the ID, $e \in \{1,\ \dots,\ |\mathcal{E}|\}$, of EC and the number of VNFs, $n \in \{1,\ \dots,\ n_s\}$, of the ordered service chain of the slice that will be placed at the chosen EC, i.e., the decision can be described as the pair $(e,\ n)$. For ensuring training efficiency, we map the above pair to the one-dimensional space by the function $f(e,\ n) = (e-1)\cdot n_s + n$. Therefore, a slice s with arrival time t_s and holding time ht_s is given to the agent at time $t_k \geq t_s$ (it may hold that $t_k > t_s$ only if the slice has been rejected at t_s. Then, the agent takes the decision $\alpha_k \neq 0$ from which we obtain $(e_k,\ v_k)$ by the inverse mapping of $f(e,\ n)$. The action α_k encodes rejection of the input slice. Next, the decision variables for $t \in \{t_k,\ \dots,\ t_k + ht_s - 1\}$ are set as follows:

$$x_{s,f}^{e_k}(t) \leftarrow 1,\ \forall f \in \{1,\ \dots,\ v_k\},$$

$$y_{s,f}(t) \leftarrow 1,\ \forall f \in \{v_k+1, \dots, n_s\},\ if\ v_k + 1 \leq n_s,$$

$$X_s(t) \leftarrow 1.$$

In case of rejection, the slice is not placed, $X_s(t_k) \leftarrow 0$ and its holding time is decreased by 1, i.e., $ht_s \leftarrow ht_s - 1$. If the new holding time is equal to zero, the slice is rejected, alternatively, the slice will be given as input to the agent at the next time slot of the control horizon, t_{k+1}.

Reward

The reward function is defined as the impact of the action to the objective function of the abovedefined optimization problem. In particular, if the action is rejection the reward is set equal to 0. Otherwise, for an action a_k that is mapped to (e_k, v_k), applied to slice s, when the agent is in state s_k, the reward is given by:

$$R(s_k, a_k) = \begin{cases} Rev_s - PC_s, & if\ slice\ s\ is\ accepted\ at\ step\ k, \\ 0, & if\ slice\ s\ is\ rejected\ at\ step\ k, \end{cases}$$

where $PC_s = PC_s^{EC} + PC_s^{RU-E} + PC_s^{E-R}$, and:

- $ReV_s = pr_s \cdot ht_s$, is the total revenue obtained by the acceptance of slice,

- PC_s^{EC} is the power consumption on the chosen EC:

$$PC_s^{EC} = \max\{ht_s - AT_{EC,e_k}(k), 0\}\gamma P^{max} + (1-\gamma)\frac{\sum_{f=1}^{v_k} c_{s,f}}{CE_{e_k}} P^{max} ht_s,$$

  where AT_{EC,e_k} is obtained from the current state s_k.

- PC_s^{E-R} is the power consumption on FH link:

$$PC_s^{RU-E} = \max\{ht_s - AT_{EC,e_k}(k), 0\}P_{net,e}^{fix} + \frac{b_{s,1}}{CB_{F,e}} P_{net}^{max} ht_s,$$

- PC_s^{E-R} is the power consumption on the MH link:

$$PC_s^{E-R} = \mathbf{1}_{\{v_k+1 \le n_s\}}[\max\{ht_s - AT_{MH,e_k}(k), 0\}\, P_{net,e}^{fix} + \frac{b_{s,v_k+1}}{CB_{M,e}} P_{net}^{max} ht_s],$$

  where $\mathbf{1}_{\{v_k+1 \le n_s\}} = 1$ if $v_k + 1 \le n_s$ and 0 otherwise.

Environment - Transition Function

The transitions are defined per RL decision step for each slice request and not per time slot, i.e., more than one updates are possible in a single time slot of the control horizon depending on the number of slices that have arrived in the corresponding slot. Thus, to clearly explain the dynamics, we define the auxiliary set of accepted slices $R_A(k)$. This set is initialized as empty, i.e., $R_A(0) \leftarrow \emptyset$ and whenever the agent takes an action $a_k \neq 0$ for a slice $s^{(k)}$ the set is updated via $R_A(k+1) \leftarrow R_A(k) \cup \{s^{(k)}\}$, otherwise $R_A(k+1) = R_A(k)$. Firstly, the slice information is updated with the information of the next slice in the queue, which is considered a stochastic transition. Regarding the network state, the computing and bandwidth capacities are updated via the equations $AC_e(k) = CE_k - C_e^k(t_k)$, $AB_{FH,e}(k) = CB_{F,e} - B_{FH,e}^k(t_k)$, $AB_{MH,e}(k) = CB_{M,E}$, where $C_e^k(t_k)$, $B_{FH,e}^k(t_k)$, and $B_{MH,e}^k(t_k)$ can be computed via the corresponding equations defined in the Problem Formulation section, by using the set $R_A(k)$, instead of $R(t_k)$, to reflect the temporary, in-time-slot, configuration. Moreover, the active time can be calculated by the equations $AT_{EC,e}(k) = \sum_{\tau=t_k}^{H-1} \max_{s \in R_A(k)} \{x_{s,1^e}(\tau)\}$ and $AT_{MH,e}(k) = \sum_{\tau=t_k}^{H-1} \max_{s \in R_A(k)} \{x_{s,1^e}(\tau) \cdot y_{s,n_s}(\tau)\}$.

Safety - Constraints Handling

All the applied decisions should respect the hard constraints of our problem. We ensure constraint satisfaction by applying action masking to violating actions [4]. Action masking is chosen as it is suitable for discrete action spaces and in the problem at hand, we can deterministically determine if an action violates a constraint.

RL-based Solution

We solve the previous MDP with the PPO algorithm. PPO is an on-policy, model-free RL algorithm and was chosen because it is aligned with the discrete action space like in our problem, requires limited hyperparameter tuning and it is compatible with the action masking mechanism.

4.3.7 Evaluation Results (RL)

<u>Evaluation setting:</u>

For the implementation of the simulation environment, we use Python version 3.10. The RL-based decision making is based on the open-source implementation of the PPO algorithm with action masking in the Python library StableBaselines3. The environment is modeled with the Gymnasium framework. Finally, the Gurobi solver and particularly, the gurobipy Python package is employed to solve offline the optimization problem for comparisons. In Table 7, we present the network infrastructure and slices parameters used for the simulations. Two types of slices are considered in our evaluations, namely, URLLC and eMBB.

Table 7. Simulation Parameters.

Parameter	**Value**
Number of ECs	3
EC, RC capacity	$64, 256$ cores
Fronthaul link capacity, delay	2Gbps, 4ms
Midhaul link capacity, delay	4Gbps, 8ms
γ	0.8
P^{max}, P^{max}_{net}	200 W
$P^{fix}_{net,e}$	160 W
Number of Requests	$U\{15, 20\}$
SFC request length	8 VNFs
CPU cores demand per VNF	$\in \{2, 4, 8\}$
Request holding time	$\min(U\{3, 6\}, 12 - t_s)$
URLLC and eMBB bandwidth requirement	100, 200 Mbps
URLLC and eMBB delay requirement	25, 50 ms
URLLC and eMBB normalized priority	3, 2.4 per time-slot
Optimization Horizon H	12 time steps

For the comparative results, two distinct baseline methods are employed. The first is the “Oracle” method, in which the above-defined optimization problem is solved under the unrealistic assumption that the future slice requests are known on beforehand and provides the optimal solution of the problem. The second baseline denoted by “RL-ST” is an RL agent similar to the one developed in the work of [6]. Contrary to our proposed RL-agent, it does not optimize slice splitting, but, instead, considers a static splitting rule chosen via experimentation. In particular, (i) URLLC slices are always split at their middle VNF and (ii) for the eMBB slices, only their second VNF is placed at an EC/DU.

Evaluation metrics:

The evaluation metrics for the performance assessment are:

1. Acceptance ratio
2. Power Efficiency
3. Objective Value

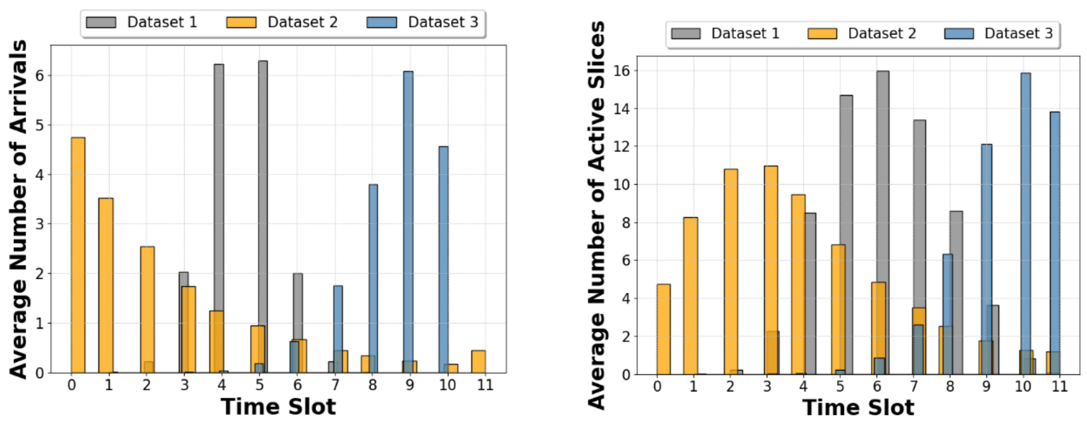
We have already defined acceptance ratio and power efficiency. Following we define the objective value metric.

Objective Value: It represents the total gain achieved by the agent at every step. It is defined as the revenue obtained from the actual slice request realization, reduced by the power consumption incurred in the resulting network state. This metric reflects the trade-off between minimizing power consumption and maximizing overall revenue, either by increasing the acceptance ratio or by prioritizing slice requests with higher priority.

It is worth mentioning that normalization of objective metrics is performed prior to simulations to mitigate disparities arising from the diverse scales of objective-related values. Two set of experiments are conducted to analyze, evaluate and compare the proposed method, which are detailed in the following subsections.

Discussion on the Results:

In the first set of experiments, we create 3 datasets with varying arrival rates per time slot in the control horizon.



(a) Average number of arrivals. (b) Average number of active slices.

Figure 4-35: Dataset information.

Specifically, arrival patterns are generated according to three distinct distributions, as visualized in Figure 4-35a: Normal distribution, $N(\frac{H}{2}-1, 0.9)$, an Exponential distribution, $Exp(\frac{4}{H})$, and a Beta distribution, $Beta(H-2, \frac{H}{6})$, rescaled to the horizon interval and grouped by time slot. The number of requests per scenario is chosen uniformly from the set $\{15, 16, \dots, 20\}$. For each distribution, 1, 000 scenarios are used for the training dataset and 10 for the test set. This experiment targets to evaluate the agent's learning ability by testing its behavior on diverse slice arrival distributions.

To begin with, we study the convergence of the training by performing 5 training instances, with different seeds, on the Dataset 1 (Figure 4-35). The moving average of the cumulative reward per episode is plotted in Figure 4-36, where we observe that after around 10, 000 episodes convergence is achieved. Similar behavior is obtained for the other datasets.

Furthermore, we train 3 models on each dataset of Figure 49. Their achieved average objective value as well as the objective value of the "Oracle" baseline method are plotted in Figure 4-37. In particular, Agent 1 represents the average of the models trained on Dataset 1, Agent 2 on Dataset 2, and Agent 3 on Dataset 3. In Figure 4-37(a), which corresponds to a test set sampled by the distribution of Dataset 1, we observe that Agent 1, which is trained on a similar dataset, performs slightly better than the other agents. In the same sense, Agent 2 performs slightly better than the other agents on a test set sampled by the distribution used for its training as depicted in Figure 4-37(b) and respectively, Agent 3 is the best performing agent for the test set corresponding to the distribution used for its training in Figure 4-37(c).

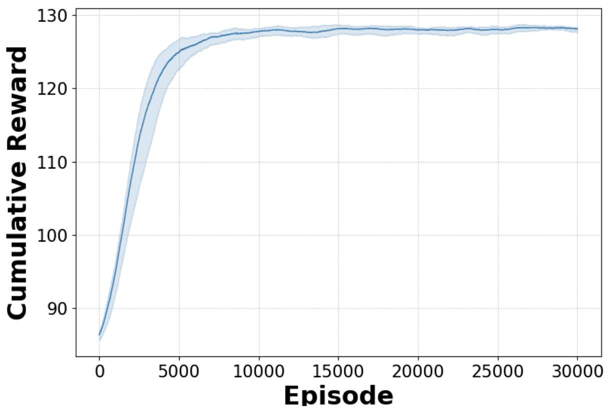
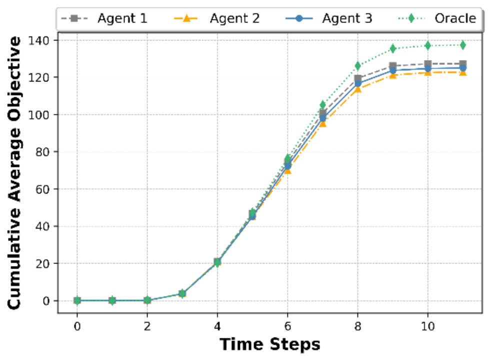
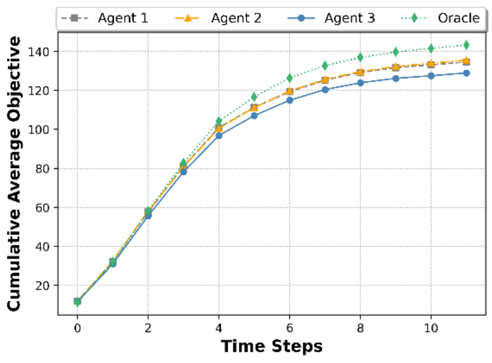
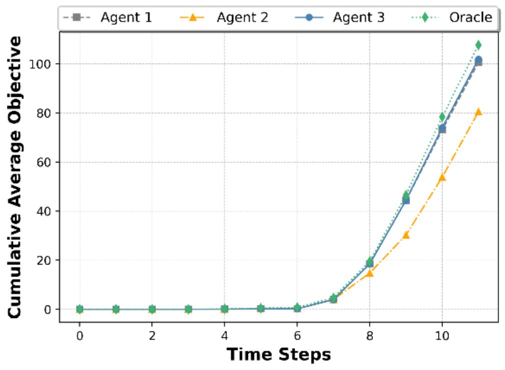


Figure 4-36: Cumulative reward evolution during training.





(a) Cumulative average objective value on test dataset 1. (b) Cumulative average objective value on test dataset 2.

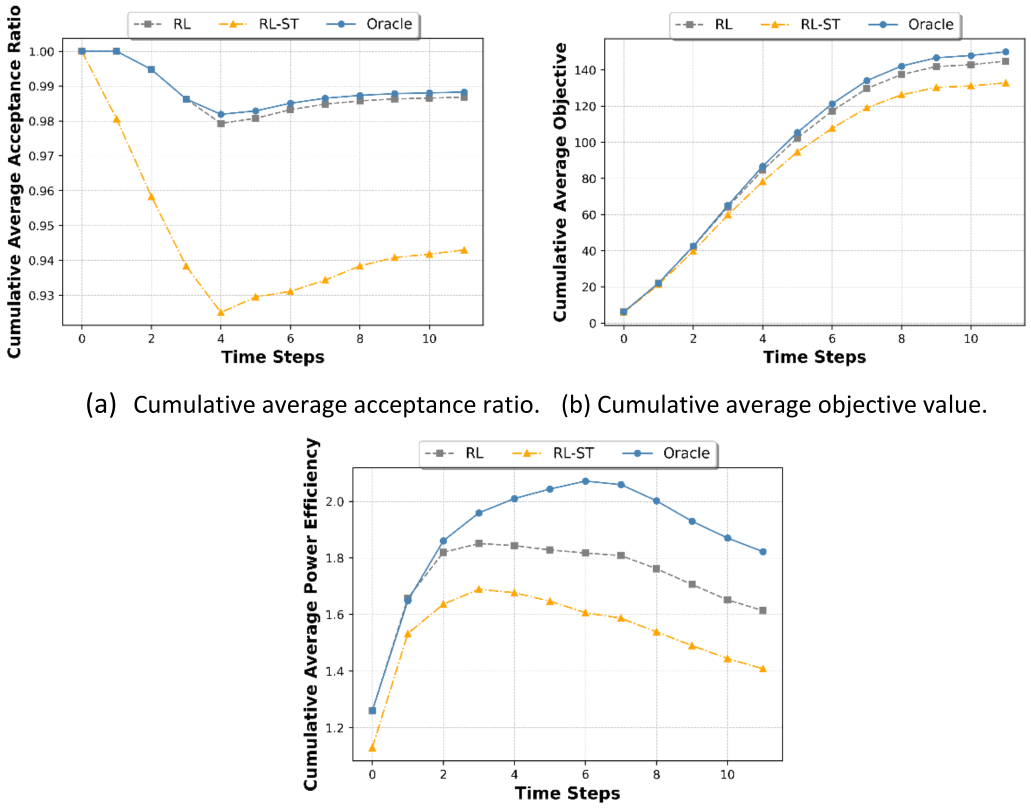


(c) Cumulative average objective value on test dataset 3.

Figure 4-37: Comparative evaluation under varying arrival patterns.

It is worth mentioning that the best performing agent for all datasets achieves an objective value close to the optimal given by the oracle. Specifically, the best performing agent achieves at least 92.5% of the optimal value. Furthermore, all agents perform well in all datasets, even on those that deviate from what they have been trained on, which indicates the good generalization possibilities of our method.

The second set of experiments is designed to assess the agent's ability to take the optimal splitting decision. To this end, the agent is compared against the "RL-ST" method, in two test scenarios with varying total number of requests. This comparison evaluates the adaptability and state-awareness of the proposed RL method. In the second experiment, we train 3 models based on our method and 3 models according to the "RL-ST" baseline to assess the importance of the dynamic splitting of slices. The Dataset 2 was used for each training instance. We create two test sets: the low load set, which corresponds to the distribution of Dataset 2, and the high load set, which follows the arrival patterns of Dataset 2, but the total number of slices is sampled from a discrete uniform distribution over the set $\{20, 21, \dots, 30\}$.

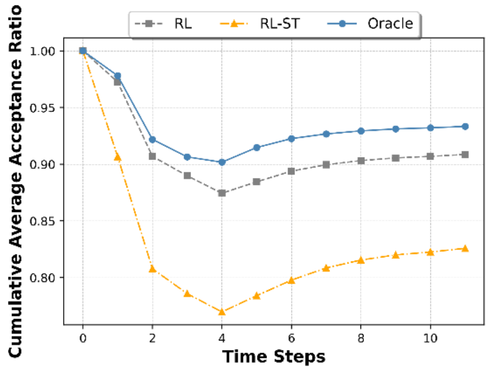


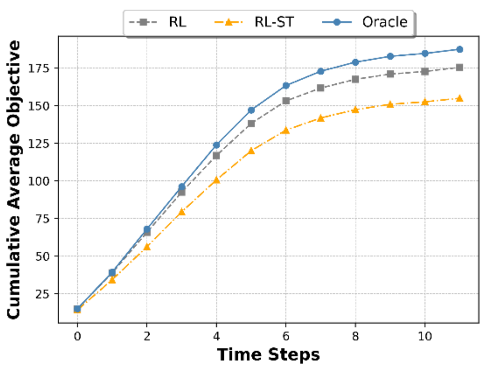
(a) Cumulative average acceptance ratio. (b) Cumulative average objective value.

(c) Cumulative average power efficiency.

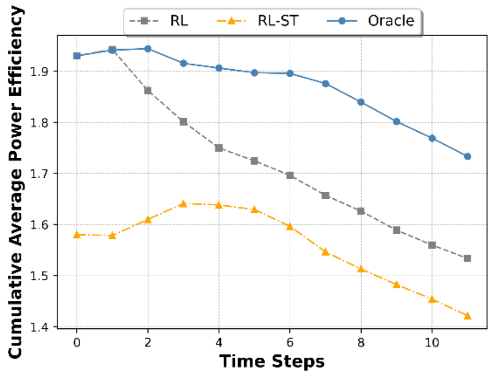
Figure 4-38: Low load conditions.

In Figure 4-38(a), we can observe that the static splitting method leads to a high number of rejections, in contrast to our method that has the ability to adjust the load between ECs and RC to achieve higher acceptance ratio. The Figure 4-38(b), (c) show that the lack of adaptability of the "RL-ST" method leads to deteriorated performance in the remaining key metrics as well. Specifically, our method shows performance gains due to dynamic splitting on average 8.14% and at maximum 16.34% with respect to the objective value metric, and on average 12.06% and at maximum 39.02% with respect to the power efficiency metric.





(a) Cumulative average acceptance ratio. (b) Cumulative average objective value.



(c) Cumulative average power efficiency.

Figure 4-39: High load conditions.

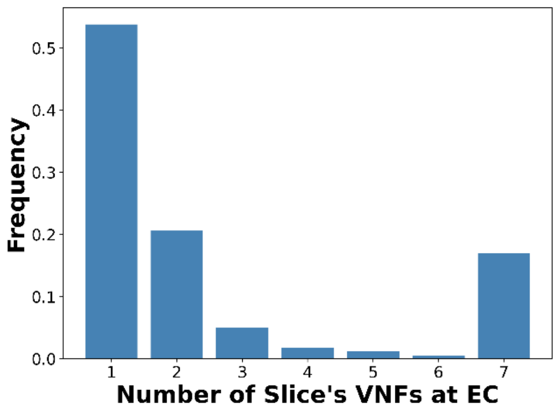


Figure 4-40: Slice splitting statistics.

The advantages of our method over the "RL-ST" method are more evident in the case of higher demand. In Figure 4-39(b), we observe that the objective value achieved by the "RL-ST" method is significantly lower than our agent, whereas our method outperforms the static splitting method also with respect to the other two assessed metrics (Figure 4-39(a), (c)). In particular, the objective value achieved by our method is on average 12.81% higher, and the average power efficiency improvement is 4.67%. In addition, the maximum observed improvement on this test dataset is 34.70% concerning the objective value and 20.86% for the power efficiency metric. In Figure 4-40, the statistics related to splitting are presented. Specifically, the frequency of each splitting decision across all scenarios of the second set of simulations is plotted. We observe that the agent tends to place only a single VNF at the EC for most slices, however, in many cases it places more than one, even the entire slice, to achieve better performance.

4.4Application Graph Deployment across Multiple Providers

4.4.1 Theoretical Foundation

In this subsection, we outline the theoretical foundations of the mechanisms developed within the Experiential Network Intelligence Function (ENIF), designed as a component for intent lifecycle management, presented analytically in [27]. These mechanisms are detailed in D3.4, where ENIF processes slice intents expressed as application graphs provided by the Business Support System Function (BSSF). In particular, we present the **Application Graph Partitioning** mechanism and the **Inter-Provider Deployment Plan** mechanism, both forming part of ENIF's Intent Provision functionality. Finally, we showcase the results of the experimental evaluation of the intent lifecycle management framework across multiple providers leveraging the simulation kit, as described in D2.4.

Application Graph Partitioning

A graph partitioning mechanism is developed to address the problem of deploying application graphs across multiple providers [27]; in this problem formulation, each application is modelled as (G_a, L_a). G_a is a connected, labelled, undirected graph (V_a , E_a); V_a is the set of the application components, and E_a is the set of relationships between them. Each application component $u \in V_a$ has a label $[c_u^L, c_u^H]$ denoting the CPU demand range; each relationship $\{u, v\} \in E_a$ is labelled by $[b_{\{u,v\}}^L, b_{\{u,v\}}^H]$, denoting the bandwidth demand range. L_a is a global label that characterizes the entire graph and models the application objective. In the current work, the application objective is high performance or energy efficiency. Let $P = \{1, \dots, p\}$ be the set of available providers. Every provider is associated with an infrastructure; this is also modelled as an undirected, labelled connected graph $G_s^p = (V_s^p, E_s^p)$, with $\{C^i\}_{i\in V_s^p}$, $\{B^{\{i,j\}}\}_{\{i,j\}\in E_s^p}$ denoting the CPU capacity on computing nodes and bandwidth on links, respectively. Additional important parameters associated with each provider include the degradation factor d_p (this factor expresses collective provider profiling and its real potential to maintain promised resources reservations, increasing as the percentage of intent violations increases), the energy consumption due to the consumption of CPU resources e_p^{CPU}, the intra-provider energy consumption due to the produced network traffic e_p^{BW}, and the inter-provider energy consumption due to the produced network traffic $e_{p,p'}^{BW}$.

All valid graph partitions of the application graph are generated for each incoming request. For each valid partition, all possible placements are tested. A candidate placement solution assigns the partition sub-graphs to the available providers. The selected providers are called to suggest a deployment plan for the assigned subparts arising from the solution of the optimization problem described below. In case of infeasibilities with respect to the providers' available resources, this candidate solution is rejected, otherwise an offering O is being formed based on the allocated CPU and bandwidth resources. Let $\{y_{u,p}\}_{u\in Va, p\in P}$ be an allocation matrix where $y_{u,p}$ is equal to 1 if the component u is deployed in the provider p, otherwise 0. Similarly, $\{cp_{u,p}\}_{u\in V_a, p\in P}$ and $\{bw_{\{u,v\},p}\}_{\{u,v\}\in E_a, p\in P}$ are defined to formulate the CPU and bandwidth allocated to a provider p for each application component and relationship, respectively. The offering varies according to the application objective.

- Performance:

$$O = \sum_{u \in V_a} \sum_{p \in P} cp_{u,p}(1 - d_p) + \sum_{\{u,v\} \in E_a} \sum_{p \in P} bw_{\{u,v\},p} + \sum_{\{u,v\} \in E_a} \sum_{p=1}^{P} y_{u,p} \sum_{p'=p+1}^{P} y_{u,p'} b^H_{\{u,v\}}.$$

- Energy efficiency:

$$O = \sum_{u \in V_a} \sum_{p \in P} cp_{u,p}(1 - d_p) e^{CPU}_p + \sum_{\{u,v\} \in E_a} \sum_{p \in P} bw_{\{u,v\},p} e^{BW}_p + \sum_{\{u,v\} \in E_a} \sum_{p=1}^{P} y_{u,p} \sum_{p'=p+1}^{P} y_{u,p'} b^H_{\{u,v\}} e^{BW}_{p,p'}.$$

It is important to note that, for the performance-related objective, when components u and v are co-located on the same physical node within a provider, the bandwidth term $bw_{\{u,v\},p}$ is set equal to $b^H_{\{u,v\}}$. Under the assumption of infinite bandwidth on inter-provider links, the bandwidth allocated between two interacting components corresponds to the upper bound when optimizing for performance and to the lower bound when optimizing for energy efficiency. Each application optimizes O over all valid graph partitions over all candidate placements. Besides the initial deployment, this mechanism is triggered by Control Loop 3 during the refinement of a single component and relocation to an alternative provider.

Intra-Provider Deployment Plan

For a single provider infrastructure, the provider $p \in P$ must solve an online placement and resource allocation problem [27]. An indicative way to formulate the problem is the following.

$$\textbf{Minimize} \sum_{u \in V_a} \sum_{i \in V_s^p} (x_u^i \cdot c_u^H - c_u^i) + \sum_{\{u,v\} \in E_a} (b^H_{\{u,v\}} - \hat{b}_{\{u,v\}})$$

$$\textbf{subject to constraints } (4.4.2) - (4.4.15) \qquad (4.4.1a)$$

$$x_u^i, f^{i,j}_{\{u,v\}}, f^{j,i}_{\{u,v\}} \in \{0,1\}, \qquad (4.4.1b)$$

$$c_u^i, b^{(i,j)}_{\{u,v\}}, b^{(j,i)}_{\{u,v\}}, \hat{b}_{\{u,v\}} \in \mathbb{N}_0, \qquad (4.4.1.c)$$

$$\forall u \in V_a, \forall i \in V_s^p, \forall \{u,v\} \in E_a, \forall \{i,j\} \in E_s^p. \qquad (4.4.1.d)$$

The placement of application components is described by binary variables x_u^i , which indicate if a component u is assigned to a computing node i. Each component is assigned to exactly one node.

$$\sum_{i \in V_s^p} x_u^i = 1, \qquad \forall u \in V_a. \qquad (4.4.2)$$

We define constraints to ensure that the assigned components will not violate the remaining CPU resources of a computing node. In contrast, the allocated resources remain within the acceptable range of the components' requests.

$$\sum_{u \in V_a} c_u^i \le C^i, \quad \forall i \in V_s^p, \qquad (4.4.3)$$

$$x_u^i \cdot c_u^L \le c_u^i \le x_u^i \cdot c_u^H, \quad \forall u \in V_a, \forall i \in V_s^p \qquad (4.4.4)$$

We establish routing variables $f_{\{u,v\}}^{(i,j)}$, $f_{\{u,v\}}^{(j,i)}$ to indicate whether an interaction $\{u, v\}$ is routed through a specific link $\{i, j\}$. Although links are modeled as undirected in each provider's infrastructure, in our problem formulation, we distinguish between the two possible directions, represented as directed links (i, j) and (j, i). These directional links share the same underlying physical resources, particularly bandwidth capacity. We impose the constraint that for each such pair, at most one direction can be selected. To ensure proper path construction and prevent the creation of loops, we augment the initial graph G_s^p by introducing a source node s that initiates all flows $\{u, v\}$, and a destination node d where these flows terminate. Each node $i \in V_s^p$ is connected to the source node s via a directed link (s, i), and to the destination node d via a directed link (i, d), both of which are assumed to have infinite bandwidth. In the following, we establish routing constraints to capture the well-known flow conservation and unsplittable flow restrictions, and constraints to prevent loops in a path. It is important to note that $\delta^-(i) = \{ (j, i) \mid \{i, j\} \in E_s^p \}$ denotes the incoming links of a node $i \in V_s^p$, assuming that the initial undirected graph G_s^p is treated as a directed graph where each undirected link is replaced by two directed links in opposite directions. Similarly, $\delta^+(i) = \{ (i, j) \mid \{i, j\} \in E_s^p \}$ denotes the set of outgoing links from node i.

$$\sum_{j\in\delta^+(i)\cup\{d\}} f_{\{u,v\}}^{(i,j)} - \sum_{j\in\delta^-(i)\cup\{s\}} f_{\{u,v\}}^{(j,i)} = 0, \quad (4.4.5)$$

$$\sum_{j\in\delta^+(i)\cup\{d\}} f_{\{u,v\}}^{(i,j)} \leq 1, \quad (4.4.6)$$

$$\sum_{j\in\delta^-(i)\cup\{s\}} f_{\{u,v\}}^{(j,i)} \leq 1, \quad (4.4.7)$$

$$f_{\{u,v\}}^{(s,i)} = x_u^i, \quad (4.4.8)$$

$$f_{\{u,v\}}^{(i,d)} = x_v^i, \quad (4.4.9)$$

$$\forall\{u, v\} \in E_a, \forall i \in V_s^p$$

Similarly to computing nodes, we define capacity constraints for the bandwidth resources of links.

$$\sum_{\{u,v\}\in E_a} b_{\{u,v\}}^{(i,j)} + b_{\{u,v\}}^{(j,i)} \leq B^{\{i,j\}}, \ \ \forall\{i, j\} \in E_s^p, \quad (4.4.10),$$

$$f_{\{u,v\}}^{(i,j)} \cdot b_{\{u,v\}}^L \leq b_{\{u,v\}}^{(i,j)} \leq f_{\{u,v\}}^{(i,j)} \cdot b_{\{u,v\}}^H, \quad (4.4.11)$$

$$f_{\{u,v\}}^{(j,i)} \cdot b_{\{u,v\}}^L \leq b_{\{u,v\}}^{(j,i)} \leq f_{\{u,v\}}^{(j,i)} \cdot b_{\{u,v\}}^H, \quad (4.4.12)$$

$$\forall\{u, v\} \in E_a, \forall\{i, j\} \in E_s^p \cup \{ (s, i) \mid i \in V_s^p \} \cup \{ (i, d) \mid i \in V_s^p \}$$

To guarantee the same bandwidth allocation at each physical link of a formed path, we introduce new integer variables $\hat{b}_{\{u,v\}}$ that denote the bandwidth assigned for the interaction $\{u, v\}$ and add the following bandwidth-conservation constraints.

$$\sum_{i\in V_s^p} b_{\{u,v\}}^{(s,i)} = \hat{b}_{\{u,v\}}, \quad (4.4.13)$$

$$\sum_{i\in V_s^p} b_{\{u,v\}}^{(i,d)} = \hat{b}_{\{u,v\}}, \quad (4.4.14)$$

$$\sum_{j\in\delta^+(i)\cup\{d\}} b_{\{u,v\}}^{(i,j)} - \sum_{j\in\delta^-(i)\cup\{s\}} b_{\{u,v\}}^{(j,i)} = 0, \quad (4.4.15)$$

$$\forall\{u, v\} \in V_a, \forall\ i \in V_s^p$$

4.4.2 Experimental Evaluation

As described in detail in D2.4, the simulation kit captures the creation of application graph services, their deployment on the multi-provider infrastructure and the continuous lifecycle management of their intent. The management framework introduces three control loops, with the first one performing semantic and syntactic validation of the client intent before the initial deployment, the second control loop models short-term intra-provider orchestration actions based on the well-being of the application component and the third control loop implements a long term intent monitoring, proposing intent refinements and inter-provider re-deployments as well as assessing the quality of each provider, useful information for subsequent deployments.

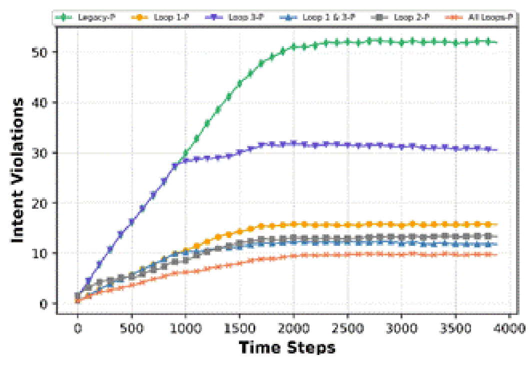
For the experimental evaluation, we assume that each application request has an aggregate workload profile (which is unknown to the control loops) of three different types: Variable, Bursty and Uniform. Apart from its type, the aggregate workload also has one out of three volume levels (Low, Medium, High) [27]. The aggregate workload is modelled as a Markov Modulated Poisson Process (MMPP), so the type of the workload determines its transition probability matrix of the next state of the internal Markov chain, and the volume sets the Poisson rate's values, based on which the aggregate workload object generates its value (following Poisson distribution) for the current slot. Furthermore, each component has also a Type (Type1, Type2, Type3) as part of the user intent which classifies the component either as CORE or SUPPORT. We assume that Type1 components are CORE and the rest are SUPPORT. This distinction is necessary for capturing the importance of each component in the application graph since CORE components are considered to be more resource demanding and in general require more CPU resources when compared with their SUPPORT counterparts. The simulation kit handles CPU and Bandwidth resources and their amount is declared through three distinct ranges (LOW, MEDIUM, HIGH), while only the CPU resource has a dynamic behavior. At each timeslot the generated CPU aggregate workload is distributed at the application components based on their own internal state and Type, and by translating through a linear function, we produce the final CPU consumption for each component for the timeslot. Concerning the infrastructure providers, each provider has a type based on which its monetization and resource quality (reduced due to oversubscription of the shared physical resources) are determined. The corrective actions of the three control loops are based on the implementation provided in MECC paper.

Two deployment scenarios are examined, the first focusing on an intent with high performance objective, identified by index P, and the second on an intent with high energy efficiency objective, identified by index E. The total number of components per application graph is considered to be up to four, while the infrastructure of each provider may reach up to ten nodes. Three infrastructure providers are considered, each of a different type; Performance-Oriented, Moderate-Cost, and Energy-Efficient. The three types offer high to low quality resources and high to low cost policies per CPU/bandwidth unit, with the inter-provider communication cost being higher that the intra-provider communication. Requests for application deployment, described by an intent of type P or type E, continuously arrive in the system.

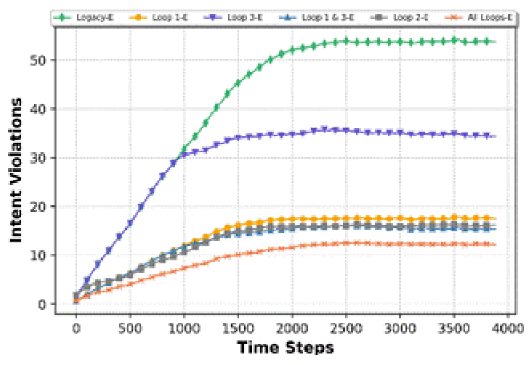
We compare the results achieved based on a Legacy Request Management (LRM) approach where no intent control loop is activated but only a mechanism that, during application runtime, initial resources reservations are attempted to be maintained at all times within the provider (a common alternative to intent control loop 2 that does not require applications to expose information to the provider), and the Intent Lifecycle Management (ILM) approaches where various control loops are activated. Six scenarios are considered, with each one activating different control loops, as follows: (i) LRM method (**Legacy**), (ii) LRM with ILM control loop 1 (**Loop 1**), (iii) LRM with ILM control loop 3 (**Loop 3**), (iv) LRM with ILM control loop 1 and 3 (**Loop 1 & 3**), (v) LRM with ILM control loop 2 (**Loop 2**), and (vi) all ILM control loops (**All Loops**). When control loop 2 is activated, violations are considered periodically within a short time interval (equal to 100 time slots) and actions are taken at the end of each period. The same short period is considered in the Legacy case as well. In the case of an activated loop 3, actions are taken after considering violations (as well as actual provided resources) for a longer period (every 1000 slots). Loop 1 takes initial corrective actions on the intent definition based on the predicted application profile. Loop 2 makes temporal adjustments within the provider to reduce violations according to the short-term application profile. Loop 3 contributes to building the actual application profile and the provider profile, driving associated actions; reconsidering intent definition and intent placement considering actual application demands and actual resources provided. The fundamental difference between Loop 2 and the considered Legacy approach is that Loop 2 uses information about violations exposed by the application, where in the Legacy case there is no insight on the application's internal operation but only an external view of the resources usage and the effort is to maintain initial reservations.

The metrics used for evaluation include the total intent violations and the percentage of intents with violations to the accepted intents for both type P and E intents, the cost incurred by deploying the applications across the providers in terms of CPU and bandwidth, and the deployed components per provider. Intent violation occurs for request i at timeslot t when at least one component’s CPU consumption is greater than the effective CPU (the actual CPU offered to the component by the provider as a result of the oversubscription). For the evaluation of the proposed framework, we examined 10 sets of providers and 5 sets of requests per provider set, thus conducting 50 experiments in total and presenting the average of the aforementioned metrics across all experiments.

In Figure 4-41, we show the total intent violations for both type P and type E intents. The LRM scenario is shown to have the poorest performance in reducing intent violations in both cases. In the type P intent (Figure 4-41(a)), significant improvement occurs in case of activation of Loop 2, since it enables dynamic resource management. Loop 1 also achieves remarkable performance by successfully profiling requests to acquire adequate resources for their execution at initial deployment. Loop 3 reduces violations on a smaller scale due to larger activation windows but if coupled with Loop 1, they outperform both Loop 1 and Loop 2 scenarios. When all loops are activated, the system achieves its best performance, as Loop 1 ensures sufficient resources for initial deployment, Loop 2 takes fast corrective actions for resource increase and Loop 3 guides initial and re-placement of type P intents at higher quality providers. In case of type E intents, a similar behavior can be observed (Figure 4-41(b)) with Loop 3 prioritizing migrations to lower cost providers. Loops 1 & 2 prove to be the most effective loops by prioritizing performance maximization and share common implementation across intent types, explaining the similar pattern in the scenarios behavior.



(a) High performance intent



(b) Energy efficient intent

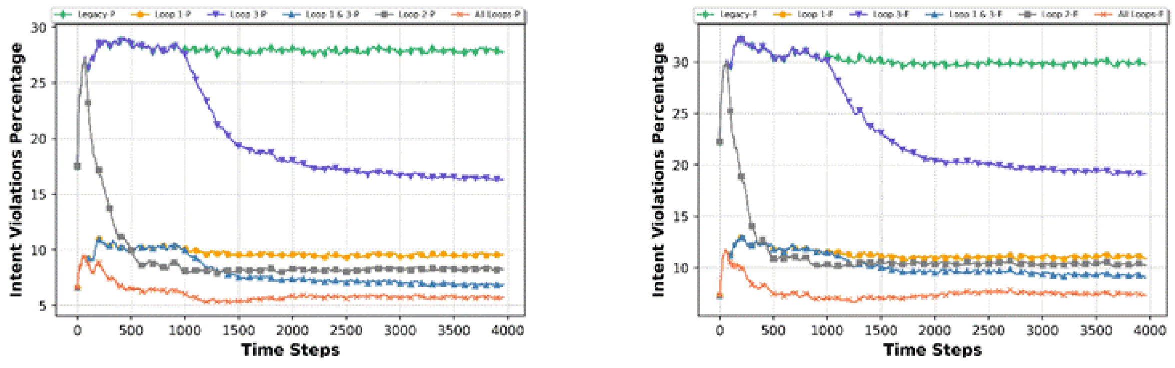
Figure 4-41: Intent Violations per time slot [27].

In Table 8, we present the amount of accepted requests for each scenario and intent type. It is clear that the LRM and Loop 3 approaches perform the best in terms of acceptance while Loops 1 & 2 exhibit the worst acceptance capability as both loops rely on greedy resource increase to reduce intent violations which leads to fast depletion of available resources for new requests. On the other hand, Loop 3, after examining an intent's workload pattern for a long time-window, it may decide to lower resources thus ensuring cost reduction and better provider availability without sacrificing performance. The All Loops scenario balances the aforementioned approaches, showcasing the trade-off between over-provisioning and provider acceptance ratio.

Table 8. Accepted requests per scenario and intent type [27].

Method	High performance	Energy efficient
Legacy	186	180
Loop 1	165	159
Loop 3	188	181
Loop 1&3	174	167
Loop 2	162	157
All Loops	171	165

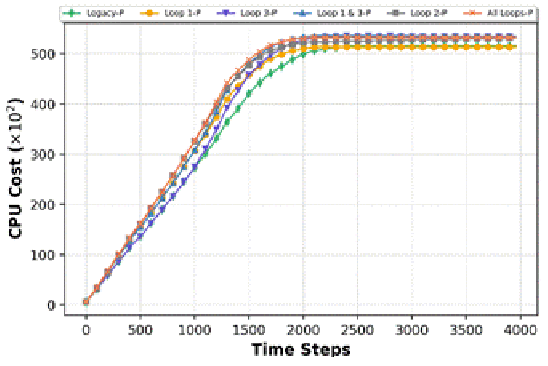
To enable a direct comparison of the lifecycle scenarios, we express their intent violations at timeslot *t* as a percentage of the total requests they have accepted up to that point. The resulting plots (Figure 4-42) confirm the insights drawn from the raw violation counts and clearly show that the All Loops scenario consistently achieves the highest quality in intent deployments. As anticipated, once the scenarios converge, the percentages for type P intents are lower than those for type E intents, illustrating the balance between guaranteeing performance and reducing cost.



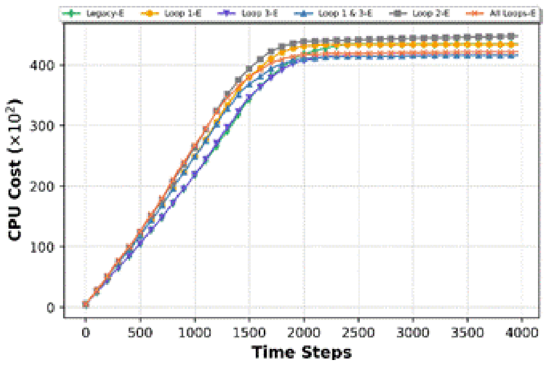
(a) High performance intent (b) Energy efficient intent

Figure 4-42: Intent Violation percentage per time slot [27].

In Figure 4-43, the CPU cost (serving as an indicator of energy usage) is notably lower for type E intents, demonstrating the effectiveness of the control loops and the initial deployment strategy under this intent directive. For type P requests, All Loops exhibits the highest energy consumption, which contrasts with the behavior observed for type E intents, where it aligns well with the goal of minimizing cost. Overall, Loop 3 is the primary driver of cost optimization, as it is the only control loop that accounts for the intent directive, resulting in higher consumption in the High-Performance context and the lowest consumption in the Energy-Efficient context.



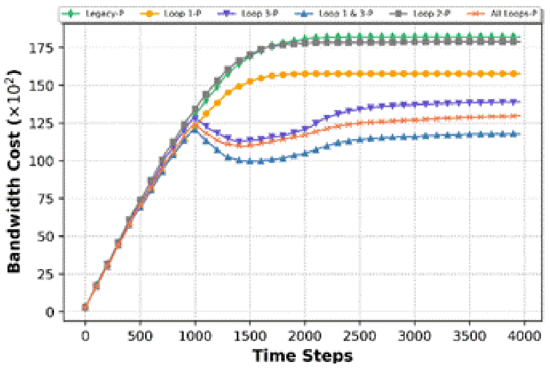
(a) High performance intent



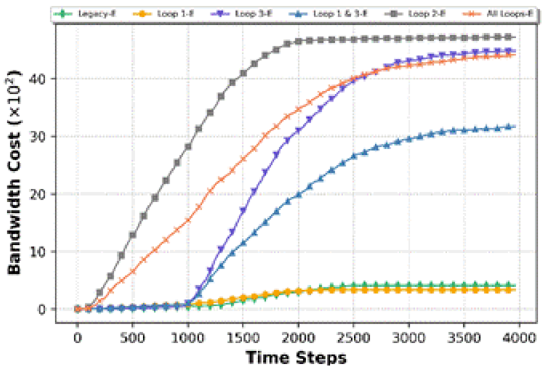
(b) Energy efficient intent

Figure 4-43: CPU cost (×10^2) per time slot [27].

In Figure 4-44, we present the bandwidth cost for each scenario. For type P intents, LRM, Loop 2 and Loop 1 present the highest bandwidth cost as their initial placement is completely random and does not consider provider cost. The scenarios that have Loop 3 activated will perform migrations that will gather many components that where mistakenly placed on lower cost providers on the higher quality providers, thus reducing their inter-provider communication. For type E intents, the bandwidth cost is significantly reduced when compared to the type P intents, as the framework prioritizes placement on the lower cost provider and ideally on the same node. When migrations have to take place (Loop 2 & 3) the deployment manager often changes the placement node of the component, thus inflicting intra-provider communication cost and in rarer cases inter-provider cost.



(a) High performance intent



(b) Energy efficient intent

Figure 4-44: Bandwidth cost (×10^2) per time slot [27].

In Figure 4-45, we show the evolution of the number of components deployed per type of provider over time for the All Loops scenario. For type P intents, before the Loop 3 activation of the first accepted request (around 1000 slots), deployment manager assumes high quality across all providers which constitutes the placement decision a random choice when the same amount of resource offering occurs. This will result in many type P components being placed on the Energy Efficient provider, where their performance will fall short. When Loop 3 updates Deployment Manager knowledge about the degradation factor of each provider, we can observe intense inter-provider migration towards Performance-Oriented Provider. For type E intents, the deployment manager is heavily inclined in placing all components to the cheaper provider if possible, which rapidly increases the number of components of the Energy Efficient provider, rapidly depleting its resources, and repeating the process with the next available low cost provider, in this case the Moderate-cost one.

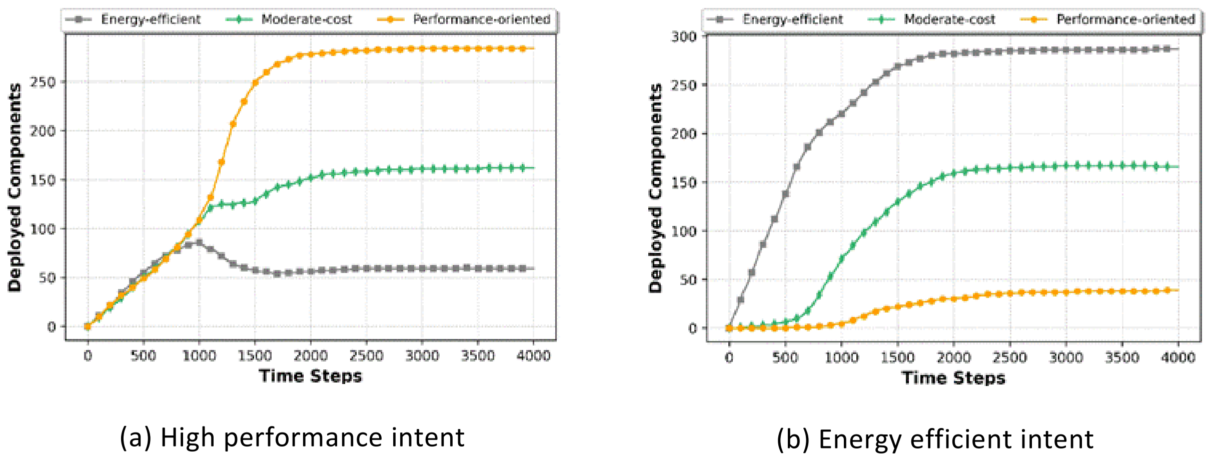


Figure 4-45: Deployed components per provider over time [27].

Overall, control Loops 1 & 2 appear to be the most effective when considering intent satisfaction, while loop 3 successfully captures the client's intent and ensures performance or cost reduction. Loop 1 relies on successful application profiling to efficiently suggest intent refinements. However, when this profiling is coarse, greedy and rule-based it can lead to over or under provisioning of resources. Loop 2 on the other hand, is a reliable mechanism that can make short-term resource adjustments based on metrics exposed to the provider about the "well-being" of the application, ensuring continuous satisfaction, with the trade-off of increasing energy consumption. Finally, Loop 3 can provide higher quality intent suggestions as well as metrics about the quality of the provider, optimizing energy consumption ideally without sacrificing performance. The whole experimentation setup, results and discussion is described in [27].

5 Green Observability and Profiling in the 5/6G Continuum

The overall 6Green observability framework is depicted in Figure 5-1. Metrics for services executed over on-premise or public infrastructures are collected through a classical Time Series Database (TSDB) solution and customized probes. Policies are used to reconcile information presented to the framework, either to extract KPIs from infrastructures metrics, either to simulate resources usage for external web services. Resulting information is provided to analytics modules and dashboards.

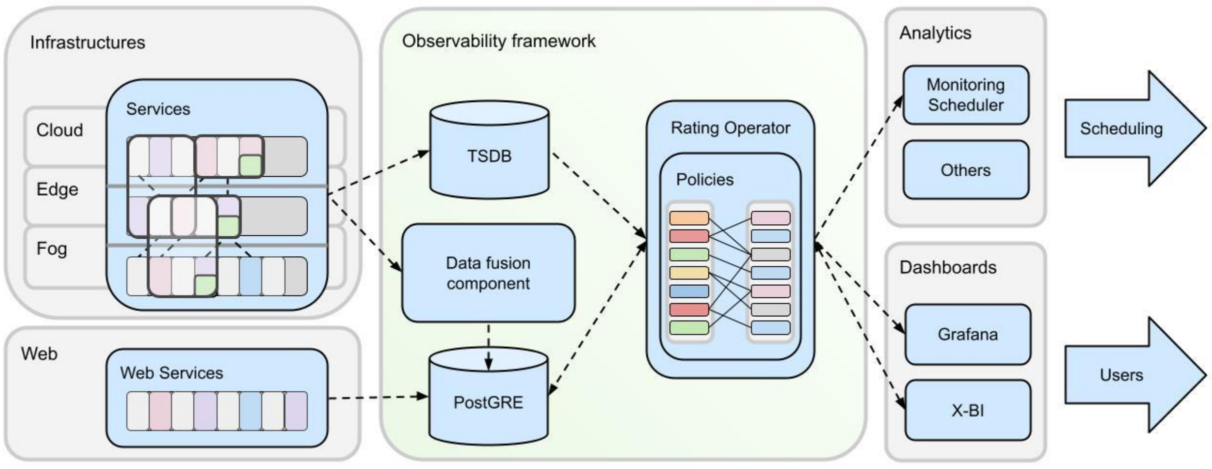


Figure 5-1: 6Green observability framework architecture.

5.1 Data Fusion Mechanisms

We consider data fusion of **observability signals** that can be classified into **metrics, logs and traces** (see Figure 5-2). **Metrics** are numerical data that capture the state of a system at a particular time or over a period. They serve as basic information for quick response and decision-making within an orchestration system, such as rule-based mechanisms for auto-scaling based on predefined thresholds. These metrics can take various forms, including counters (e.g., counting incoming HTTP requests), gauges (e.g., measuring the current depth of a queue), or histograms (e.g., depicting the duration of a request). Examples of commonly monitored metrics encompass resource usage (like CPU or memory usage), traffic volume (such as incoming or outgoing traffic per second), and the number of requests handled (e.g., HTTP requests served per second). Monitoring components within cloud and edge computing orchestration platforms typically provide access to a wide array of such metrics [28].

Logs are structured records of individual events, presented in a textual format that humans can readily understand. They typically detail usage patterns, events, activities, and operations within an orchestration system, such as application debug or error messages. By aggregating data from multiple logs, valuable insights into specific situations or events can be gleaned. Third-party tools, compatible with cloud and edge computing orchestration platforms, often facilitate access to these logs [28].

A **distributed trace** encompasses a sequence of operations that represent a unique transaction managed by an application. Consequently, these traces can be correlated with a request's scope. Each transaction or request comprises a series of operations spanning across the microservices of the application. By analyzing distributed traces, we can gain better insights into the events occurring during a distributed transaction and pinpoint any delays or bottlenecks within the overall process. Common examples of insights provided by distributed tracing include latencies in software execution within microservices, interactions between microservices, and end-to-end latencies for fulfilling specific requests. Third-party tools, with varying degrees of integration and interoperability, typically provide access to distributed tracing information within cloud and edge computing orchestration platforms [28].

Observability involves integrating various types of signals, including metrics, logs, and traces. When deciding which signals to monitor, it's important to balance the richness of information available against considerations of performance and complexity. Once the appropriate set of signals has been identified, collecting relevant information relies on properly instrumenting the deployed software.

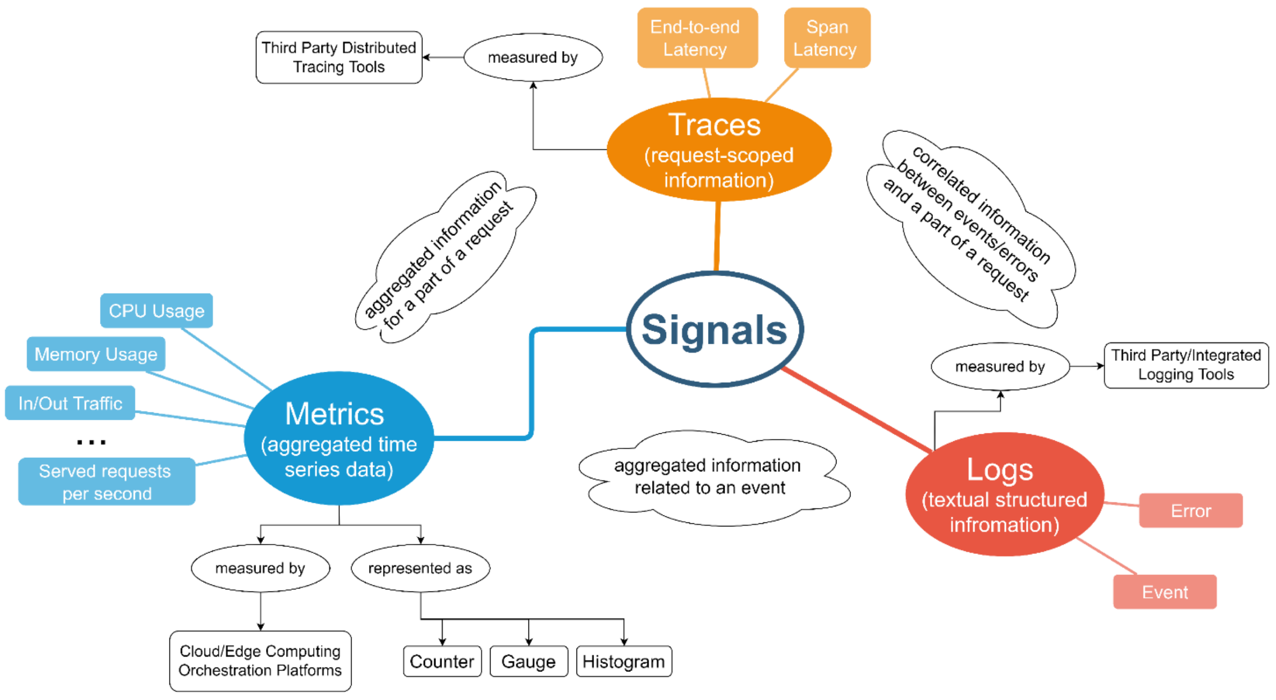

Third Party Distributed Tracing Tools
measured by
End-to-end Latency
Span Latency
Traces (request-scoped information)
aggregated information for a part of a request
correlated information between events/errors and a part of a request
Signals
CPU Usage
Memory Usage
In/Out Traffic
...
Served requests per second
Metrics (aggregated time series data)
measured by
Third Party/Integrated Logging Tools
aggregated information related to an event
Logs (textual structured infromation)
Error
Event
measured by
represented as
Cloud/Edge Computing Orchestration Platforms
Counter
Gauge
Histogram

Figure 5-2: Classification of signals into metrics, logs and traces [28].

In order for a data model to be able to support orchestration in the compute continuum, it needs to support 2 main aspects:

- a representation of the resources, i.e., a model of the computing and the network infrastructure,
- an application graph representation, modelling designed service communication patterns and data transfers.

Taking into consideration a multi-cluster infrastructure, the main entities constituting the computing representation should include the cluster and the corresponding physical or virtual computing nodes of each cluster. The network representation is described as a graph of network connections (links) between network nodes (e.g. routers/switches), while it is also crucial to provide support for virtual links created by network controllers (e.g. SDN) to make network performance guarantees (Figure 5-3). Runtime information of applications placed in the continuum is important to be registered close to the infrastructure for supporting orchestration actions. Specifically, a deployment identifying the placement cluster and replication factor of each service represents the application instance, while an individual runtime instance records a replica's resource utilization and state.

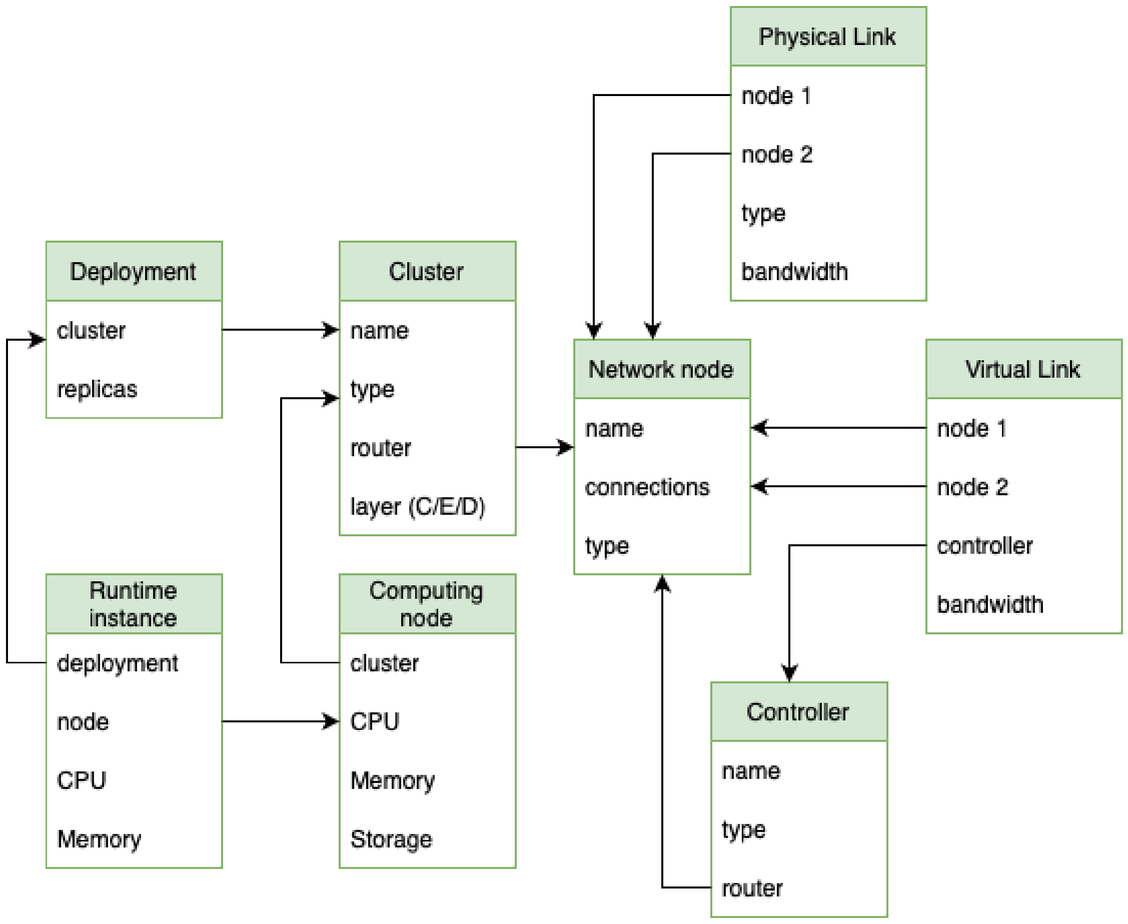


Figure 5-3: Data representation.

A detailed application description is important for identifying its underlying complexity and its connectivity characteristics that can be utilized to optimize its performance (Figure 5-4). An application graph is defined as a network of services interacting with each other through their endpoints. For one service to access another, a service call (link) is made having a specific payload size. A sequence of service calls makes up a directed acyclic graph (DAG), a workflow executing a certain functionality. The service's runtime information is recorded in the infrastructure model as discussed above, so each service maps to a deployment.

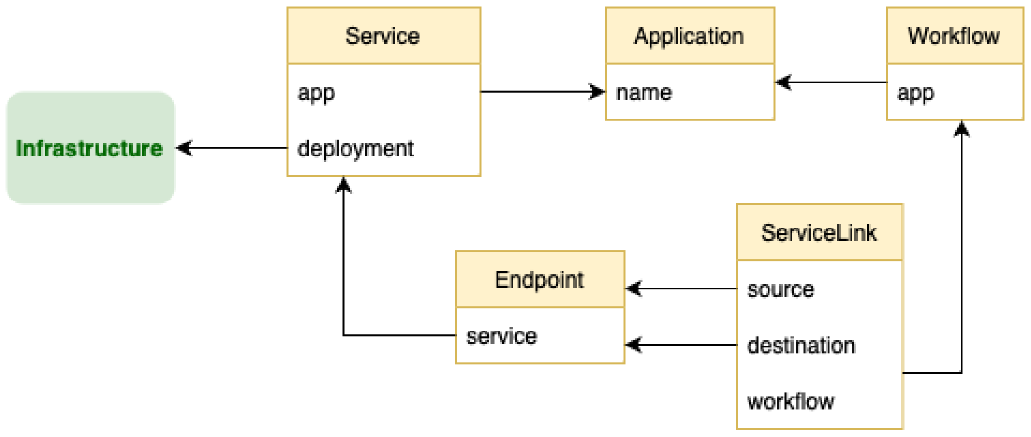


Figure 5-4: Application/Service representation.

5.2 Profiling Mechanisms

A recent and comprehensive study published by CNIT [29], has clearly summarized the profiling power consumption most used approaches in the following Figure 5-5, where we schematically see the most interesting aspects:

- Workload characterization and instrumentation.
- Resource instrumentation.
- Resource Specific workloads.
- Direct power measurements.
- Temperature (e.g., CPU package temp, Fan Speed etc.).

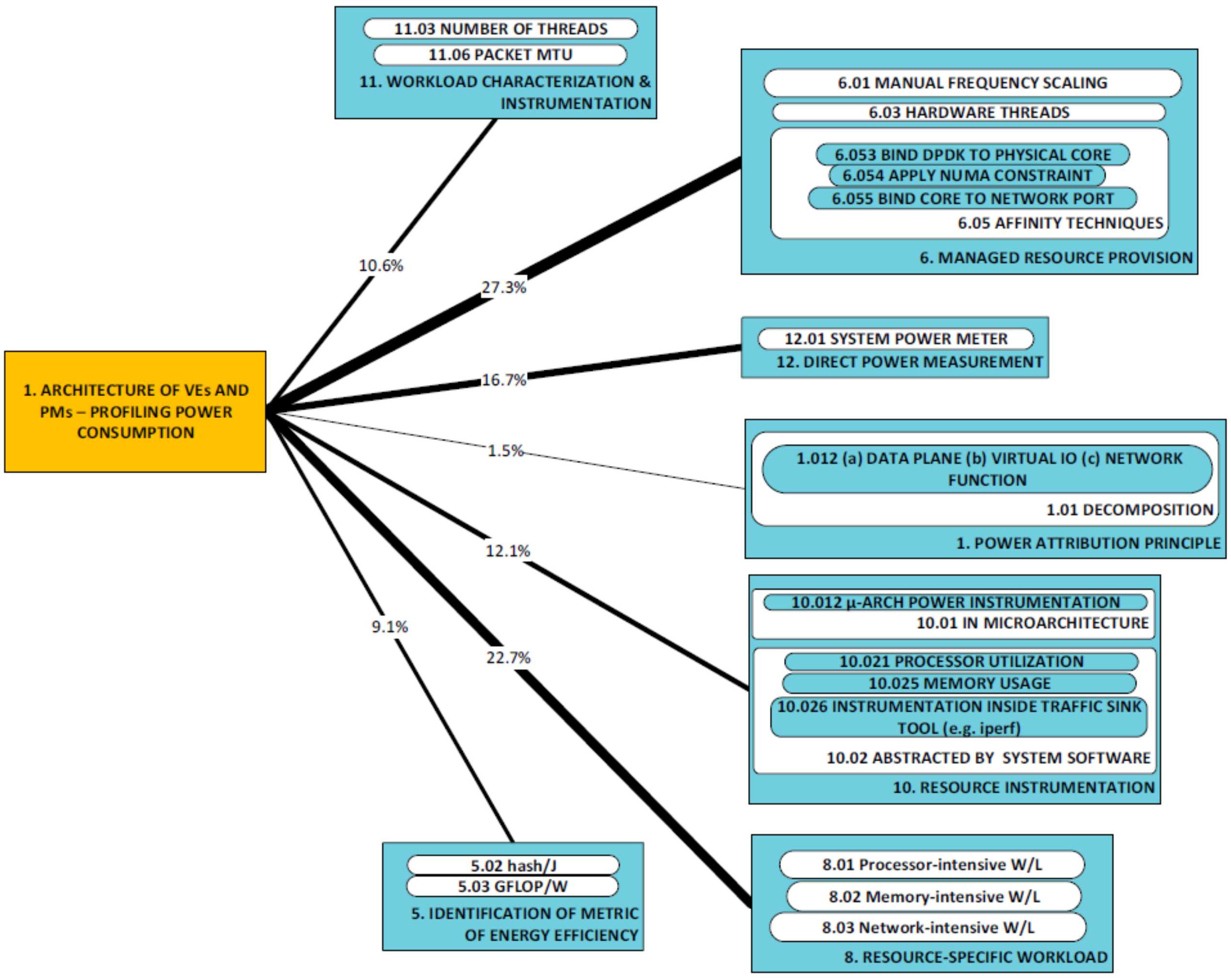

11.03 NUMBER OF THREADS
11.06 PACKET MTU
11. WORKLOAD CHARACTERIZATION & INSTRUMENTATION
6.01 MANUAL FREQUENCY SCALING
6.03 HARDWARE THREADS
6.053 BIND DPDK TO PHYSICAL CORE
6.054 APPLY NUMA CONSTRAINT
6.055 BIND CORE TO NETWORK PORT
6.05 AFFINITY TECHNIQUES
6. MANAGED RESOURCE PROVISION
10.6%
27.3%
12.01 SYSTEM POWER METER
12. DIRECT POWER MEASUREMENT
16.7%
1. ARCHITECTURE OF VEs AND PMs – PROFILING POWER CONSUMPTION
1.5%
1.012 (a) DATA PLANE (b) VIRTUAL IO (c) NETWORK FUNCTION
1.01 DECOMPOSITION
1. POWER ATTRIBUTION PRINCIPLE
12.1%
9.1%
22.7%
10.012 μ-ARCH POWER INSTRUMENTATION
10.01 IN MICROARCHITECTURE
10.021 PROCESSOR UTILIZATION
10.025 MEMORY USAGE
10.026 INSTRUMENTATION INSIDE TRAFFIC SINK TOOL (e.g. iperf)
10.02 ABSTRACTED BY SYSTEM SOFTWARE
10. RESOURCE INSTRUMENTATION
5.02 hash/J
5.03 GFLOP/W
5. IDENTIFICATION OF METRIC OF ENERGY EFFICIENCY
8.01 Processor-intensive W/L
8.02 Memory-intensive W/L
8.03 Network-intensive W/L
8. RESOURCE-SPECIFIC WORKLOAD

Figure 5-5: Profiling Power Consumption.

5.2.1 The Rating Operator Tool

The Rating Operator serves as a Kubernetes application. Operators can be considered as extensions to the microservices hypervisor. The Rating Operator follows this approach to extend the native API. Functionally, it enables the transformation of metrics into customizable Key Performance Indicators (KPIs) and provides interfaces for their use in monitoring, supervision or other purposes. This multi-tenant, configurable, and lightweight operator addresses users’ rating needs. Our involvement in the 6Green Project focused on enhancing the tool's ability to model metric transformation rules as units, previously implemented directly in the source code. In the context of the transformation logic we support, metrics are sourced from various time series databases (TSDBs), encompassing a wide range of business values with fine diversity and granularity. In response, we opted to enhance the efficiency of TSDB queries by introducing a dynamic approach, consolidating and optimising them. This same principle was extended to the values propagated in these TSDBs queries.

The Rating Operator tool facilitates the transformation of metrics into Key Performance Indicators (KPIs), allowing users to define rules for this conversion. For instance, it empowers users to convert metrics related to compute resources into KPIs representing energy consumption, carbon footprint, or pricing. Figure 5-6 also illustrates an example of a Rating Operator use case of carbon emission calculation from energy consumption (orange part). In this example, illustrating the API developed as part of the project, a first template (query, in blue) provides a dynamic query, second templates (values, in pink) supply the values populating the query. Finally, the system returns respective objects (instances, in green) reporting the metric(s) transformation. Orange part illustrates an example of a Rating Operator use case of carbon emission calculation from energy consumption.

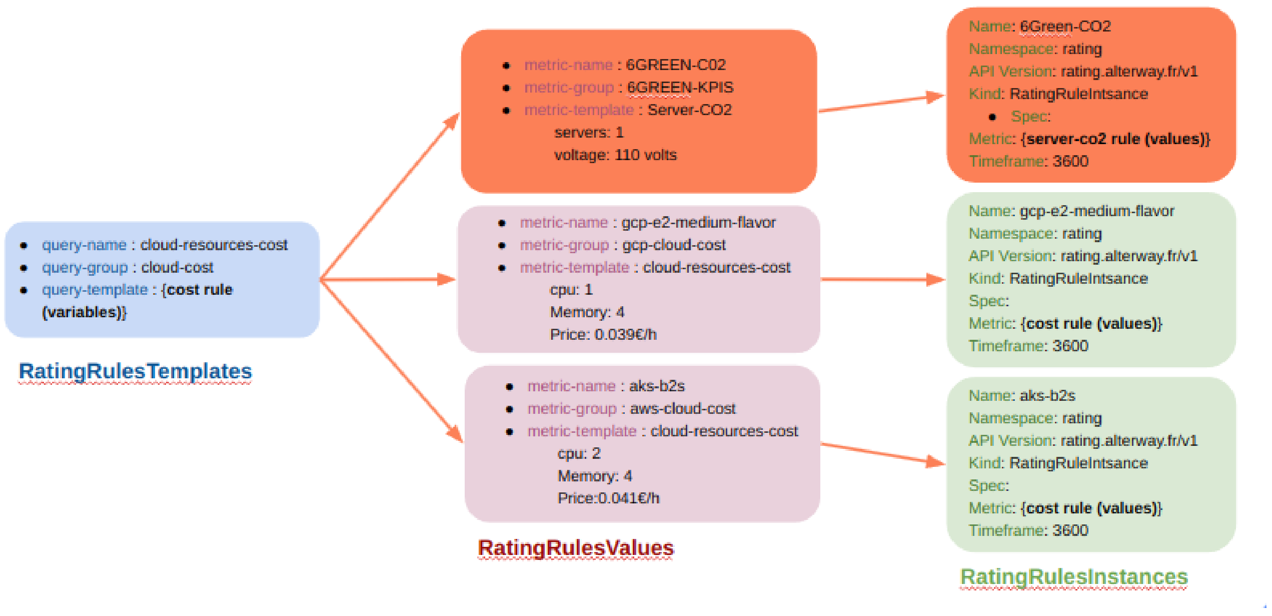

query-name : cloud-resources-cost
query-group : cloud-cost
query-template : {cost rule (variables)}
RatingRulesTemplates
metric-name : 6GREEN-C02
metric-group : 6GREEN-KPIS
metric-template : Server-CO2
servers: 1
voltage: 110 volts
metric-name : gcp-e2-medium-flavor
metric-group : gcp-cloud-cost
metric-template : cloud-resources-cost
cpu: 1
Memory: 4
Price: 0.039€/h
metric-name : aks-b2s
metric-group : aws-cloud-cost
metric-template : cloud-resources-cost
cpu: 2
Memory: 4
Price:0.041€/h
RatingRulesValues
Name: 6Green-CO2
Namespace: rating
API Version: rating.alterway.fr/v1
Kind: RatingRuleIntsance
Spec:
Metric: {server-co2 rule (values)}
Timeframe: 3600
Name: gcp-e2-medium-flavor
Namespace: rating
API Version: rating.alterway.fr/v1
Kind: RatingRuleIntsance
Spec:
Metric: {cost rule (values)}
Timeframe: 3600
Name: aks-b2s
Namespace: rating
API Version: rating.alterway.fr/v1
Kind: RatingRuleIntsance
Spec:
Metric: {cost rule (values)}
Timeframe: 3600
RatingRulesInstances

Figure 5-6: Example of a Rating Operator use case.

The Rating Operator tool is oriented towards providing a versatile solution for metrics transformation at different architectural levels. By offering this capability, it becomes a pivotal component in the ecosystem, enabling the exposure of aggregated metrics and Key Performance Indicators (KPIs) directly to applications. This strategic approach significantly reduces the reliance on centralised metrics collection and mitigates concerns related to the data volume resulting from metrics collection. The core of this concept is the empowerment of each architectural layer to define and generate metrics relevant to its specific functions. This decentralisation of metrics transformation ensures that applications can access and utilise tailored metrics and KPIs without the need for a centralised authority. This not only streamlines the integration of metrics into application logic but also enhances the overall energy efficiency of the system. Furthermore, by allowing metrics transformation at various architectural levels, the Rating Operator contributes to a more distributed and responsive system. Applications can dynamically adapt to changing conditions by utilising locally transformed metrics, leading to a reduction in the latency associated with centralised metrics collection. This, in turn, enhances the real-time nature of the data available to applications, fostering agility and responsiveness.

In the realm of efficient metrics management, the utilisation of Custom Resource Definitions (CRDs) within the Rating Operator tool proves to be a game-changer. Specifically, when applied to remote servers and services, CRDs empower users to finely tune and customise the configuration of resources, thereby enabling precise control over the pace of metrics retrieval. Users can define tailored configurations for each remote server or service, outlining the parameters governing metrics retrieval intervals. This granular control is important, especially in diverse and dynamic environments where different servers or services may have distinct requirements for metric update frequencies. Moreover, by applying scheduling to CRDs, users can dynamically reconfigure the metrics retrieval pace based on evolving needs or changing conditions. This adaptive approach ensures that the system optimally adjusts to varying workloads or operational demands, enhancing overall efficiency. For example, during periods of high demand or critical activities, users can increase the frequency of metrics retrieval for specific servers or services. Conversely, during less critical times, they can schedule a more relaxed pace to conserve resources and minimise unnecessary data transfer. This capability not only optimises resource utilisation but also contributes to the responsiveness and adaptability of the system.

Figure 5-7 illustrates the use case of metrics transformation at various architectural levels. The observation of the metric update pace can be leveraged to define new CRDs configuration to limit collection pace in regards to these updates frequencies.

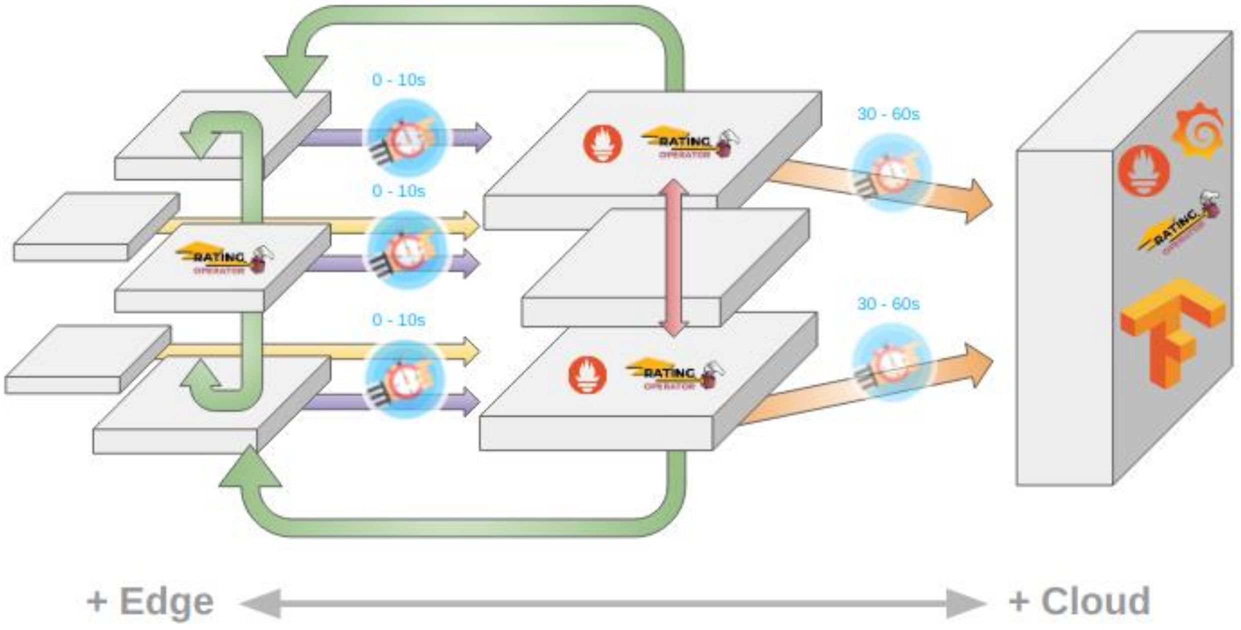


Figure 5-7: Illustration of Rating Operator providing metrics transformation at various architectural levels.

In addition to these features, we recently developed the Rating Operator API, which is also used to expose metrics to users. This API offers a structured and flexible interface to interact with transformed metrics. Endpoints are organized by their respective resources (e.g., namespaces, pods), and follow a consistent grammar across categories.

Below is an example demonstrating a simple query to an endpoint that requires no parameters:

```
$ curl http://127.0.0.1/namespaces
{
   "results":[{"namespace":"kube-system","tenant_id":"default"},{"namespace":"longhorn-system","tenant_id":"default"},{"namespace":"monitoring","tenant_id":"default"},{"namespace":"rating","tenant_id":"default"},{"namespace":"unspecified","tenant_id":"default"}],"total":5
}
```

Another example uses URL parameters. Endpoints of this type are labeled as [*URL*]. In the example below, we use the */metrics/<metric>/<aggregator>* pattern. The aggregator handles the time range, and parameters are passed via the URL. Here, the '*daily*' aggregator is used:

```
# We use  the 'daily' aggregator for the example.
$ curl http://127.0.0.1/metrics/co2-simulation-eu/daily
{"results":[{"value":15.07968}],"total":1}}
```

Monitoring Pace Scheduler

Traditional system monitoring often depends on fixed scraping intervals, where metrics are collected at predetermined, constant times (e.g., every **x** seconds or x minutes). Although straightforward to implement, this method presents several drawbacks, especially in dynamic environments or user-centric applications with fluctuating activity levels. Fixed intervals can lead to inefficient resource utilization, as data is continuously collected even when there are no meaningful updates, consuming unnecessary computation and network resources. This inefficiency becomes more pronounced when monitoring metrics that rarely update. Moreover, in highly dynamic systems, fixed intervals might miss capturing critical changes promptly, leading to gaps in important data. In resource-limited environments, the constant overhead from fixed scraping can further strain system performance without offering additional value when the system is stable. Traditional methods also lack flexibility, making them less suitable for adaptive, workload-sensitive monitoring.

To address these limitations, we introduce the **Monitoring Pace Scheduler**, a system that dynamically adjusts scraping intervals based on the observed rate of metric updates. When metrics are stable, the scraping frequency is reduced to minimize redundant data collection. Conversely, during periods of rapid change, the frequency is increased to ensure accurate and timely data capture. This adaptive strategy improves overall resource efficiency—reducing computation, network load, and storage—making it highly suitable for scalable and dynamic environments. A configurable threshold allows users to balance between monitoring precision and resource efficiency, enabling flexibility depending on application needs and metric behavior.

Figure 5-8 shows the result of applying this dynamic monitoring approach. Data points collected under thresholds of 0.1% and 0.5% demonstrate that the overall shape and trends of the metric are preserved, while the number of collected points is reduced compared to a baseline fixed-interval method. This reduction underscores the system's ability to adapt scraping intervals, preserving data fidelity while optimizing resource usage.

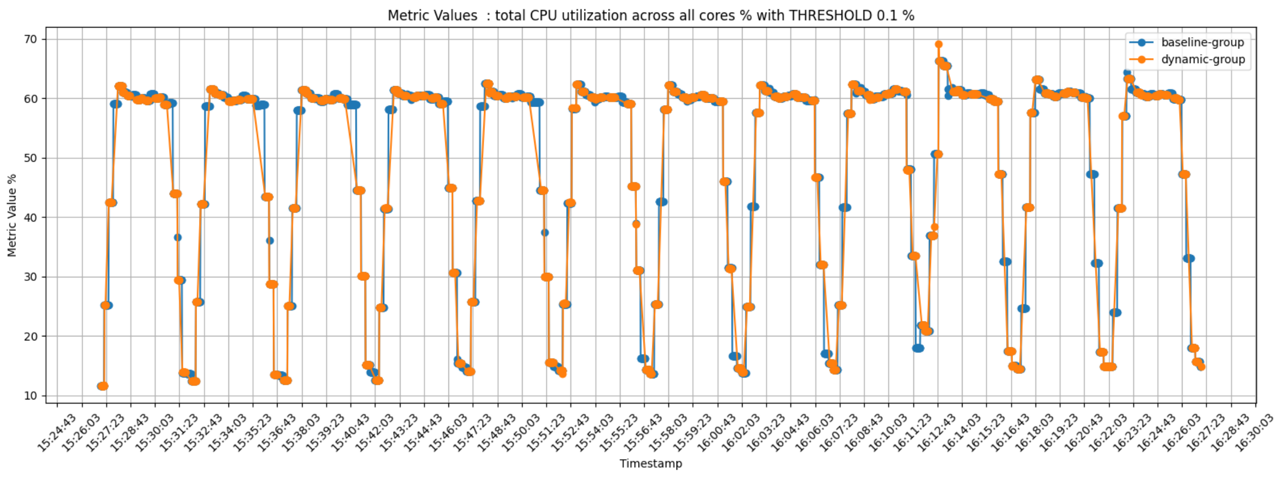


Figure 5-8: Total CPU utilization (%) with 0.1% threshold, influencing data collection frequency.

Table 9: Comparison of the baseline and dynamic groups with different thresholds.

Groups	Median scrape interval (s)	Precision (%)	Overlay_dx	MAPE (%)	Bandwidth Reduction (%)	Network Traffic Reduction (%)	Storage Reduction (%)
Baseline	15	-	-	-	-	-	-
Dynamic (0.1%)	20	75.39	0.956	3.48	34.5	34.6	49.5
Dynamic (1%)	46	73.32	0.9473	4.67	34.6	34.7	49.5
Dynamic (10%)	36	67.76	0.9377	4.74	34.9	34.9	49.5
Dynamic (80%)	80	60.00	0.8084	24.5	37.6	37.6	50.5

5.2.2 Resource Profiling Related to Elasticity and Resource Efficiency

Resource autoscaling is a key characteristic of network management systems that wireless network operators are using in order to provide highly reliable, low latency, large-scale networking services. 5G networks are the recent answer to tackle this growing networking demand. One of the key approaches is deploying network services in a cloud-native environment.

In case of containerized network services, Kubernetes provides a threshold-based solution for dynamic scaling, Horizontal Pod Autoscaler (HPA[18]). For a given time t_i and a performance metric, HPA calculates the required number of replicas $P_{desired}\ [t_i\ ]$ based on:

$$P_{desired}\ [t_i\ ] = \left\lceil P\ [t_i\ ] \cdot \frac{M\ [t_i\ ]}{M_{desired}} \right\rceil$$

[18] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

where current replica count denoted as $P\,[t_i]$, $M[t_i]$ denoted as the current metric value and $M_{desired}$ is the desired threshold for the selected metrics. This solution is reactive and HPA takes scaling actions only if the static threshold is met. When a scaling action is triggered, there is a time delay for creating and making a new replica operational, which might affect the QoS based on the resource demand at that time.

An alternative solution for Kubernetes HPA is proposed in [30], where an AI-assisted proactive scaling solution is developed, that can balance the trade-off between operational cost and QoS for a CNF deployment in a cloud-native environment. To be proactive, the proposed solution utilizes a multi-variant multiple steps time series forecasting Gated Recurrent Unit (GRU) neural network-based model that predicts the future resource consumption of the pods which belongs to a single deployment. The dynamic scaling is calculated by dynamic thresholding rules that utilize the predicted metric values provided by the forecasting model.

Solutions leveraging Artificial Intelligence (AI), Machine Learning (ML), and Data Analytics (DA) show great promise in delivering significant advancements in 5G and beyond complex network environments. By utilizing those technologies, we can propose innovative mechanisms for analysing and predicting network behavioural patterns, and extract profiles associated with the resource requirements of network services. In [31] an application profiling modelling component is introduced, that collects a set of distinct resource profiles, all associated with a resource allocation solution that calculates scaling decisions based on a simple ML technique, without violating QoS requirements, in case of Kubernetes Edge Clusters. This ML technique involves the classification of distinct combinations of computing resources concerning the application's service rate.

In 6Green we could follow the approach of [32], where an integrated framework based on open-source tools is proposed, that offers flexibility in service providers to realise experiments and achieve profiling of their applications in terms of resource and elasticity efficiency. The benchmarking describes the process of running experiments. After the benchmarking process, the resulting data is kept in a time-series database from where it can be picked up for the profiling process.

A set of analysis processes are supported to extract insights [32] such as:

- Resource efficiency analysis for the identification of resource consumption trends and capacity limits, used for planning optimal reservation of resources. The considered monitored metrics combine a resource usage metric (e.g. CPU usage, memory usage) with a service output metric (e.g., traffic served, HTTP requests served, active users). Such an analysis is realised through the production of (multiple) linear regression models.
- Elasticity efficiency analysis to assess the performance of scaling operations, along with the impact of scaling actions in the service output efficiency (e.g., traffic served by a VNF). Elasticity efficiency is expressed as a pair of discrete metrics, namely application capacity change (incremental capacity change related to a scaling action) and capacity change lead time (time required for a capacity change). Such an analysis is primarily based on monitoring and visualisation of elasticity actions. In a second stage, training and application of machine learning models for automated elasticity actions enforcement is considered by service providers, facilitating the undertaking of proactive elasticity actions for guaranteeing QoS.
- Correlation analysis for the identification of strong and statistically significant correlations among infrastructure and VNF-specific metrics, leading to various insights (e.g., which parameters are highly dependent, which parameters can create bottlenecks in the overall performance). Such an analysis is realised through correlograms.
- Forecasting based on time-series decomposition mechanisms. Such mechanisms are applied over resource usage or workload metrics and provide feedback to elasticity efficiency mechanisms. Various forecasting models are supported based on the type of the time series data.

- Graph analysis for identification of bottlenecks in software functions' calls and the consideration of software updates for optimal service provision. Such an analysis is valuable for software consisted of microservices, where performance issues and bottlenecks due to software functions' calls can be identified and provided as feedback to software developers.

5.3 Estimation of Energy Consumption of Hardware Components

Estimating the embodied energy consumption of virtualized components (i.e. containers and VMs) is challenging, primarily because hardware resources are not reserved for single virtual components. In 6Green, MDAF and IDAF are utilized to map hardware power consumption effectively.

Scaphandre[19], which relies on Intel RAPL[20] counters and on the time spent on each process by CPU to compute the power consumption per process (containers and VMs), traditionally measures only direct power consumption of virtual components at the CPU and, only in some cases, DRAM controller levels.

However, it is noted that indirect contributions arise from processes necessary to maintain servers and virtualized environments. Our approach extends Scaphandre by incorporating "embodied" power consumption, defined as power usage from kernel-level processes not directly attributed to containers or pods hosting only 6G components. To appropriately distribute this embodied power among virtual components, we leverage metrics from cAdvisor[21], which provides resource (CPU, memory, network, etc.) utilization for containers. The mapping of the **"embodied"** power consumption is shown in Figure 5-9. The kernel-level metrics are divided, manually, into categories based on affinity. Then, each category is proportional ascribed to the appropriate virtual components. Regarding Kubernetes containers, that are expected to be the main form under which a 6G network will be deployed, we exploit the cAdvisor metrics: the purple, light blue and orange categories are mapped proportionally to the CPU, network, and memory consumption, respectively. While for VMs and Docker containers, supposing their resource utilization are not available, the mapping is uniform among all the instances. Finally, the monitoring category is isolated since it includes all the processes, not containerized, that are needed for monitoring purposes.

5.3.1 Energy Consumption Measurements Based on Kubernetes, Scaphandre and Kepler

To demonstrate our approach, we deployed a containerized Iperf application in a 2-node Kubernetes cluster. The setup consisted of one master node (4 CPUs, 264 GB RAM) and one worker node (2 CPUs, 96 GB RAM), both equipped with Mellanox MT27500 NICs. On both servers, monitoring applications are deployed. The tests consist of one couple of Iperf3 server and client deployed as K8s containers: one in each K8s node in order to generate inter-node traffic. UDP traffic is generated while the bitrate of Iperf3 is changed from 1 Gigabit/s to 1Mbit/s. Each generation lasts 15 minutes and is followed by a 5-minute pause. The monitoring applications mentioned before export the metrics on to a Prometheus[22] database with a 10 second scraping interval.

Figure 5-10 shows the power consumption of the hosts and the Iperf containers. The green plot represents the power consumption of the whole server (i.e., including every component) measured by the Raritan power outlet (the IX7™ PDU Controller in detail). The orange plot represents the power consumption of the whole server measured by Intel RAPL. The yellow plot shows the power consumption of the Iperf containers (i.e., the server and the client in Figure 5-10), the grey plot represents the power consumption of the Iperf containers produced by the MDAF; finally, the blue plot represents the different bit rates which the test used.

[19] https://hubblo-org.github.io/scaphandre-documentation/
[20] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html
[21] https://github.com/google/cadvisor
[22] https://prometheus.io/

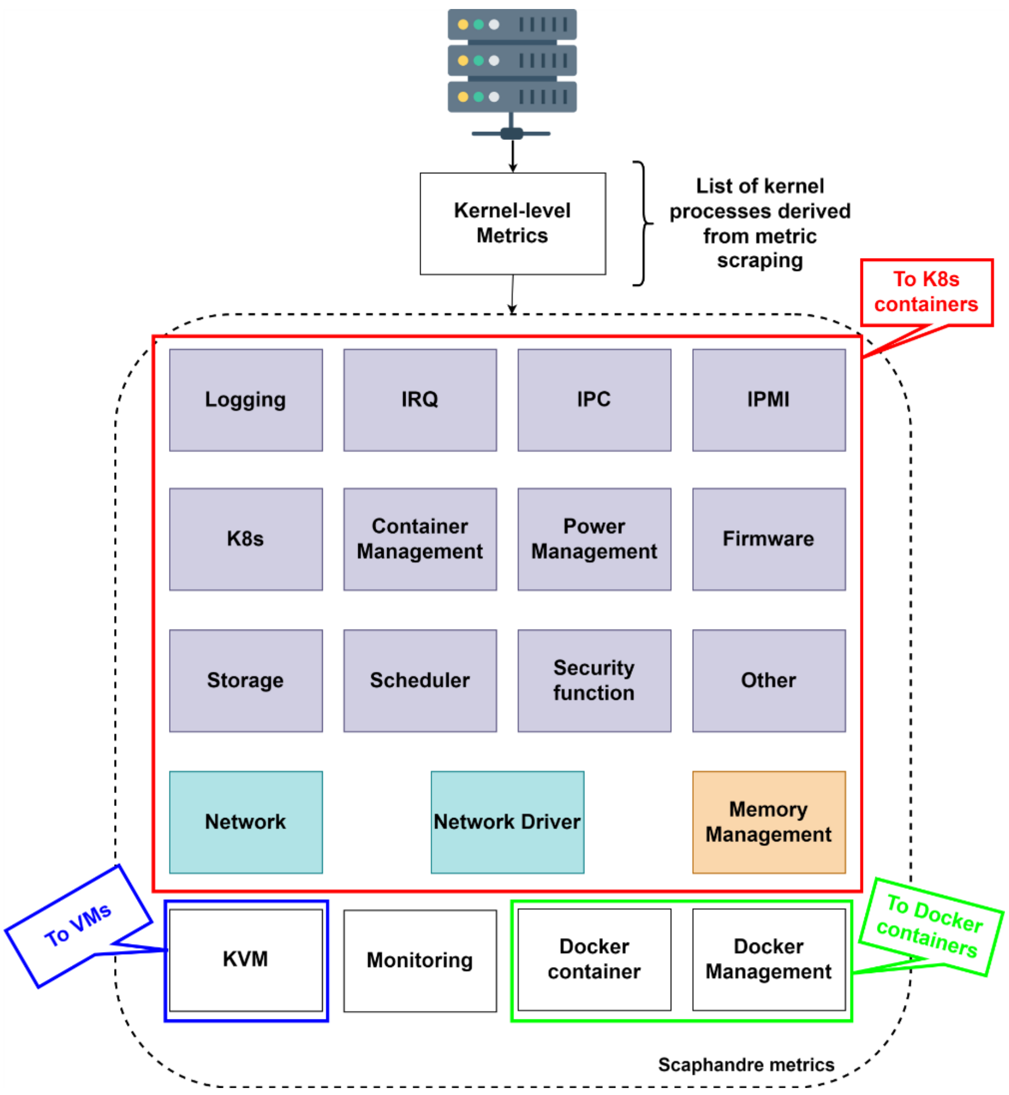


Figure 5-9: Categorization and mapping of the kernel level metrics.

First, let us focus on the server-level results: in both figures the Raritan power consumption is higher than the RAPL one; this is to be expected since the RAPL component only measures the power consumption of the CPU, while the Raritan one considers every component including the power supply. Then, considering the grey and yellow plots, it is worth noticing that, as expected, the container power consumption plots have lower values with respect to the other two.

Additionally, a comparison between the proposed solution (i.e., MDAF) and the Scaphandre solution is needed. In Figure 5-10 we can notice that Scaphandre underestimates the container power consumption since it considers only the container direct usage of the CPU. While our proposed MDAF takes into account the "embodied" power consumption due to all the kernel processes needed to keep the whole system (virtualization platforms included) up and running. Moreover, as shown in Figure 5-10, both the server and the client power consumption (i.e., yellow and grey plots) decrease with the Iperf Bitrate. This is more visible in the server than in the client. The former processes receiving packets when an interrupt is launched and then sends back reply packets. While the latter simply generates only the first packet and then sends it according to the decreasing bitrate. Finally, comparing the two sides of Figure 5-10 (i.e., the Iperf server and client), it can be noticed from the yellow and grey plots that the client consumes much less power than the server.

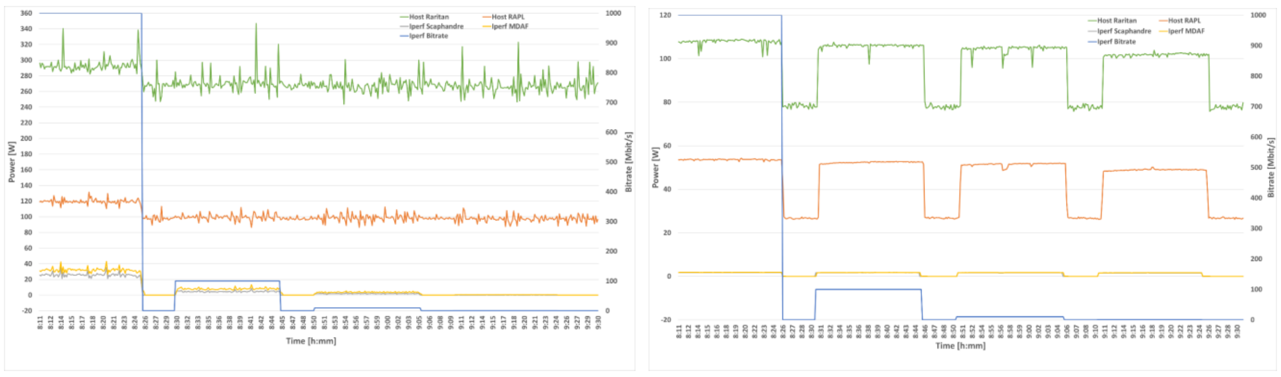


Figure 5-10: Power consumption of both the host (left) measured by the Raritan and RAPL and of the Iperf server/ client (right) measured by Scaphandre and the MDAF while the Iperf bitrate varies from 1 Gigabit/s to 1 Mbit/s.

Furthermore, to point out more easily the differences between the MDAF and Scaphandre plots, Table 10 is inserted. It shows the difference in % (in terms of mean value and standard deviation) between the MDAF and the Scaphandre container power consumption for the Iperf server and client respectively. First, analyzing the server columns of Table 10, we can notice that the difference between MDAF and Scaphandre increases when the Iperf bitrate decreases. Therefore, as the throughput decreases, the operations per packet the server carries out increase. This occurs because when the incoming traffic volume is low, an interrupt is launched every one or very few packets; while, when the incoming traffic volume is high, an interrupt is launched every several packets. Then, comparing the server and client columns, it can be noticed that, on the one hand, the mean value of the difference in the client case is almost constant, while this is not true for the server; on the other hand, the standard deviations are much lower in the client (and almost constant) case compared to the server case. This shows a much higher variability in the server rather than the client. This can be explained with the tasks that each does. The server's variability is much higher because its tasks (processing packets and sending replies) depend on external triggers (interrupts caused by incoming packets). While the client consists in a software that generates and sends packets; therefore, the operating system scheduler (rather than an external trigger) is in charge of reserving the CPU for the Iperf software. This results in less variability in the client.

Table 10: Mean value and standard deviation of the difference in % between the MDAF and the Scaphandre container power consumption for the Iperf server and client respectively.

Iperf Bitrate	Mean value (server)	Std (server)	Mean value (client)	Std (client)
1 Gbit/s	19.6%	2.01%	4.35%	1.35%
100 Mbit/s	43.4%	4.31%	3.86%	1.34%
10 Mbit/s	52.0%	7.40%	4.06%	0.88%
1 Mbit/s	56.7%	8.40%	3.92%	1.08%

Following, we provide results regarding the mapping of consumption of hardware components to cloud resources (Kubernetes setup).

Energy measurement tool comparison between Scaphandre and Kepler

Having deployed a Kubernetes single node cluster on bare metal, we have installed Kepler and Scaphandre together in order to perform a comparison between both tools. We have used Prometheus to scrape both tools' metrics every 10s and configured Grafana in order to create a visual dashboard where it could be seen, in several visualizations, the details of the different metrics. One of the main differences between Kepler and Scaphandre is that Scaphandre gives most of its energy metrics in Gauge type and in terms of Power(W), while Kepler in Counter type and in terms of energy(J). So, when comparing one to the other we need to convert the energy metric to power using Promql's rate operator. In all the figures we have Scaphandre on the left and Kepler on the right. First of all, we have the node power consumption in Watts (Figure 5-11).

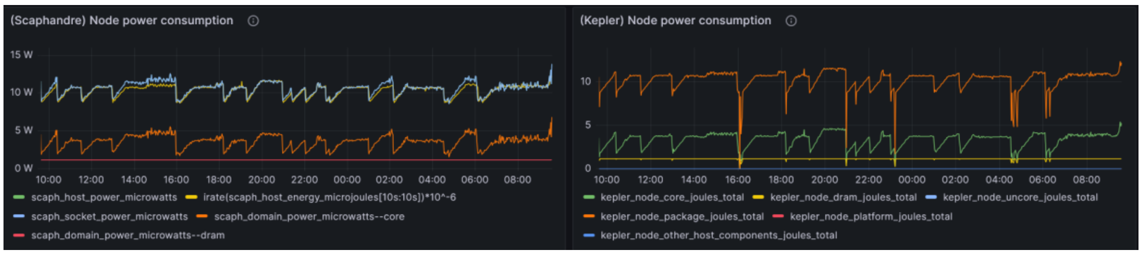


Figure 5-11: Node power consumption.

Then, host power vs aggregation of processes (Figure 5-12). This represents the previous metric compared directly with the sum by node of the containers' metrics.

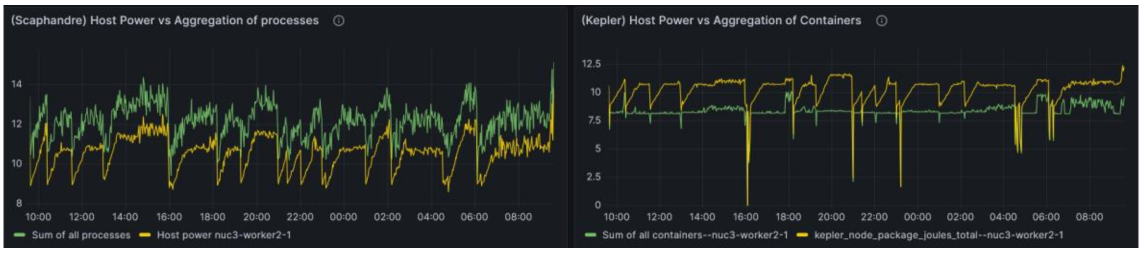


Figure 5-12: Host power vs Aggregation of processes/containers.

In Figure 5-13, Figure 5-14 and Figure 5-15 we can see directly the metrics of the containers of the host in different types of visualizations: Time series, table and state timeline.

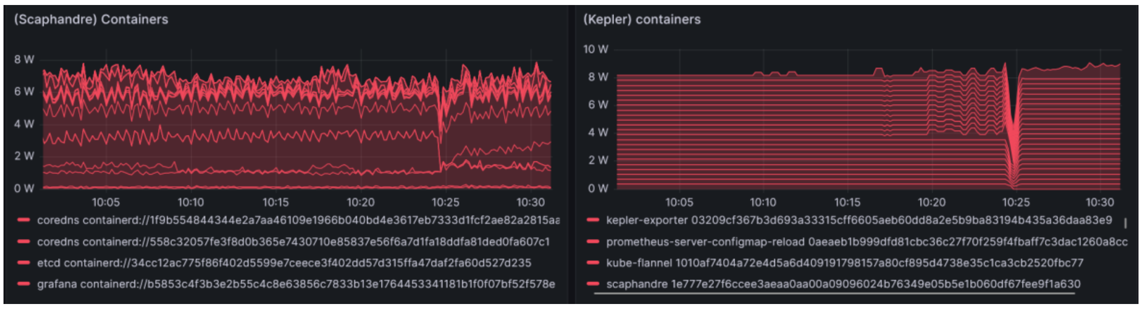
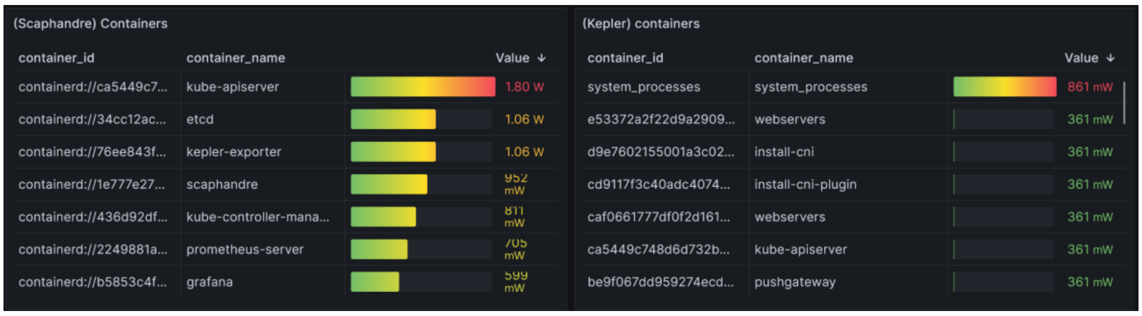


Figure 5-13: Containers.



(Scaphandre) Containers

container_id	container_name	Value ↓
containerd://ca5449c7...	kube-apiserver	1.80 W
containerd://34cc12ac...	etcd	1.06 W
containerd://76ee843f...	kepler-exporter	1.06 W
containerd://1e777e27...	scaphandre	952 mW
containerd://436d92df...	kube-controller-mana...	811 mW
containerd://2249881a...	prometheus-server	705 mW
containerd://b5853c4f...	grafana	599 mW

(Kepler) containers

container_id	container_name	Value ↓
system_processes	system_processes	861 mW
e53372a2f22d9a2909...	webservers	361 mW
d9e7602155001a3c02...	install-cni	361 mW
cd9117f3c40adc4074...	install-cni-plugin	361 mW
caf0661777df0f2d161...	webservers	361 mW
ca5449c748d6d732b...	kube-apiserver	361 mW
be9f067dd959274ecd...	pushgateway	361 mW

Figure 5-14: Containers (table).

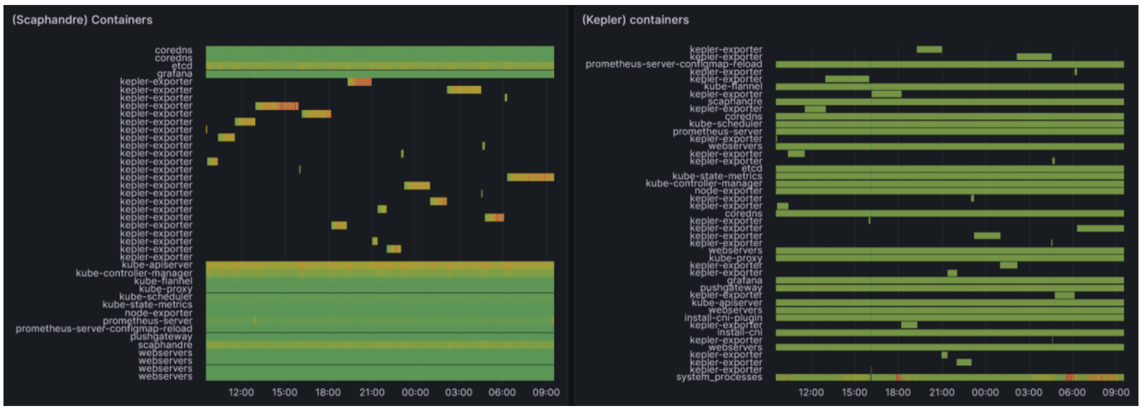


Figure 5-15: Containers (state timeline).

Lastly, in Figure 5-16 we have the metrics aggregated by pod.

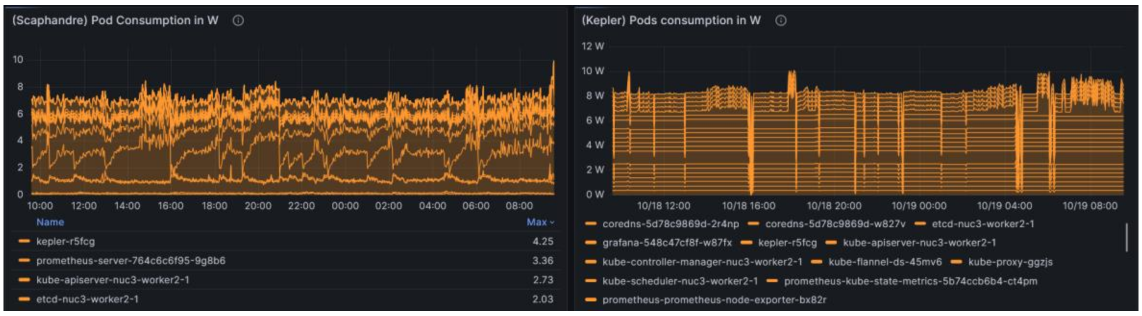
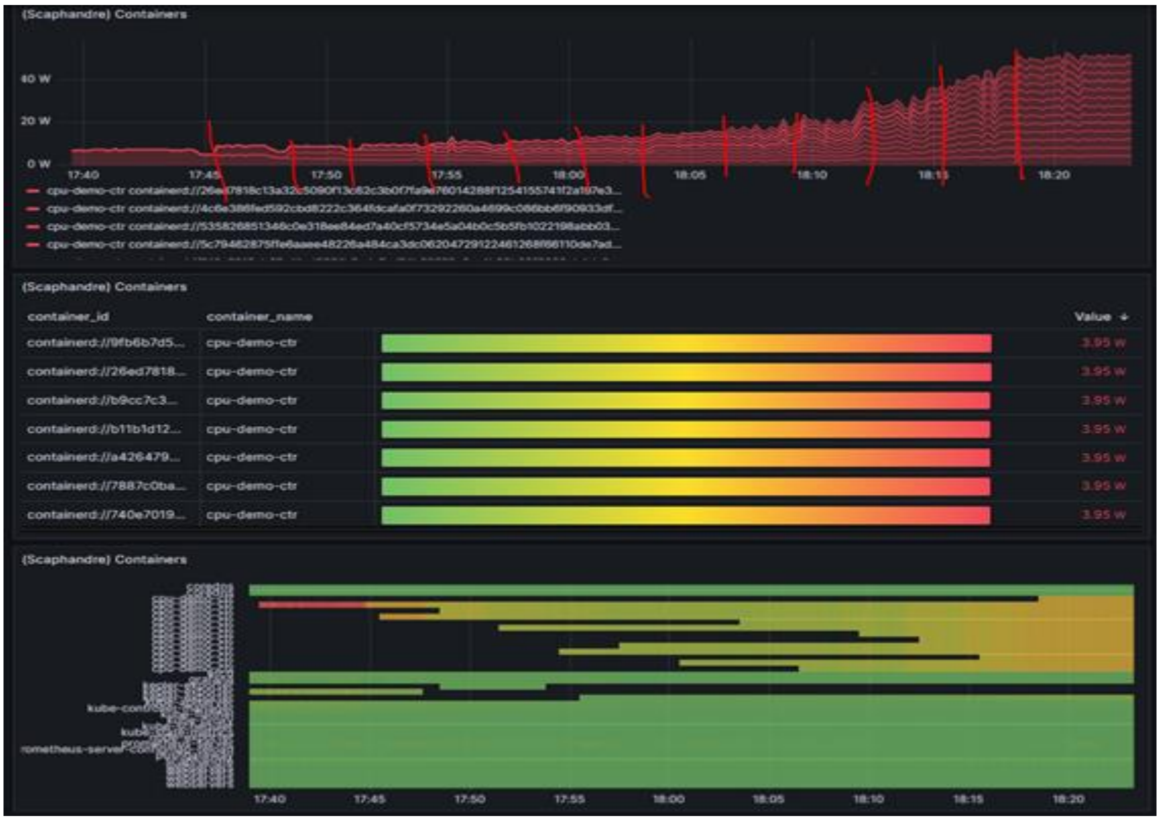


Figure 5-16: Pods.

In Figure 5-17, we can see the results of a test that we performed. The test consists in horizontally scaling a pod that request 0.5 Cores. We trigger the scaling every 3 minutes. The results are in our opinion quite normal, although interesting. We can see that, up to a certain point, scaling horizontally causes the consumption/container to decrease (even though the total consumption goes up). However, when certain load is reached, the addition of more replicas causes the total consumption to grow exponentially and the share per container to increase.

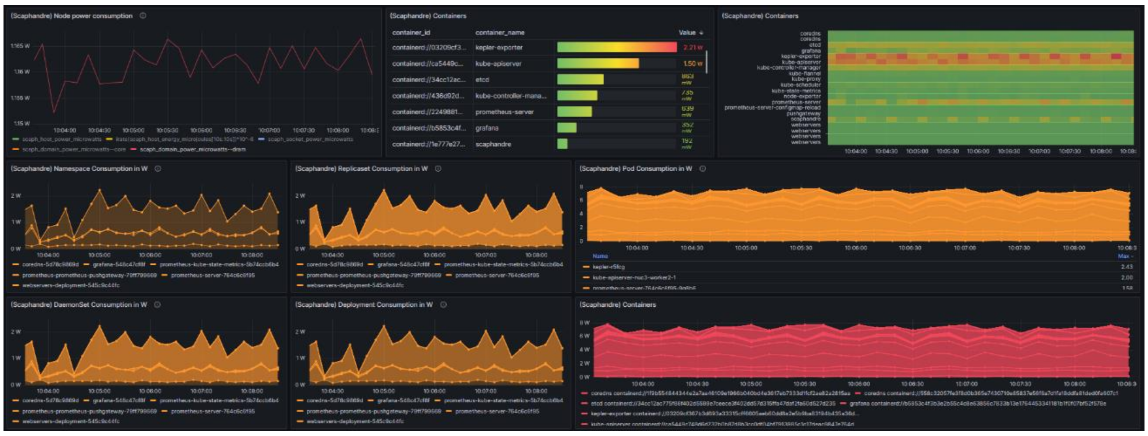

(Scaphandre) Containers
40 W
20 W
0 W
17:40
17:45
17:50
17:55
18:00
18:05
18:10
18:15
18:20
cpu-demo-ctr containerd://26ed7818c13a32c5090f13c62c3b0f7fa9e760142881125415574112a1197e3...
cpu-demo-ctr containerd://4c6e386fed592cbd8222c364fdcafa0f73292260a4699c086bb6f90933df...
cpu-demo-ctr containerd://5358268513 46c0e318ee84ed7a40cf5734e5a04b0c5b5fb1022198abb03...
cpu-demo-ctr containerd://5c79462875ffe6aaee48226a484ca3dc06204729122461268f66110de7ad...
(Scaphandre) Containers
(Scaphandre) Containers
coredns
kube-con
kube
rometheus-serve

container_id	container_name	Value
containerd://9fb6b7d5...	cpu-demo-ctr	3.95 W
containerd://26ed7818...	cpu-demo-ctr	3.95 W
containerd://b9cc7c3...	cpu-demo-ctr	3.95 W
containerd://b11b1d12...	cpu-demo-ctr	3.95 W
containerd://a426479...	cpu-demo-ctr	3.95 W
containerd://7887c0ba...	cpu-demo-ctr	3.95 W
containerd://740e7019...	cpu-demo-ctr	3.95 W

Figure 5-17: Horizontal pod scaling test.

Kubernetes Consumption Measurement Using Scaphandre

Using Grafana, we created a dashboard to visually analyse the consumption of a Kubernetes cluster broken down into the different resources that such cluster might have, such as: Containers, Pods, Deployments, Replicasets, DaemonSets and Namespaces (Figure 5-18).

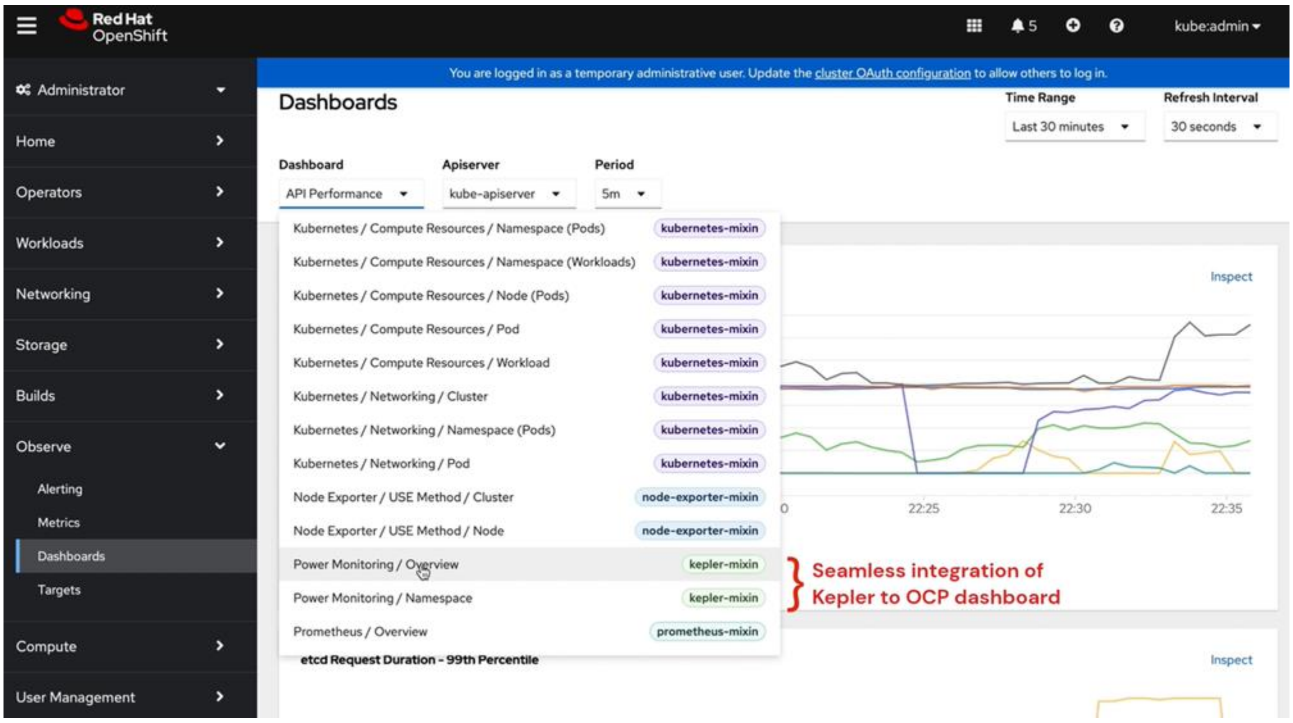

(Scaphandre) Node power consumption
(Scaphandre) Containers
(Scaphandre) Containers
(Scaphandre) Namespace Consumption in W
(Scaphandre) Replicaset Consumption in W
(Scaphandre) Pod Consumption in W
(Scaphandre) DaemonSet Consumption in W
(Scaphandre) Deployment Consumption in W
(Scaphandre) Containers
10:04:00
10:04:30
10:05:00
10:06:00
10:07:00
10:08:00
kepler-r5fcg
coredns-5d78c9869d
grafana-548c47cf8f
prometheus-kube-state-metrics-5b74ccb6b4
prometheus-server-764c6cf95
webservers-deployment-545c9c44fc

container_id	container_name	Value
containerd://03209cf3...	kepler-exporter	2.21 W
containerd://ca5449c...	kube-apiserver	1.50 W
containerd://34cc12ac...	etcd	863 mW
containerd://436d92d...	kube-controller-mana...	735 mW
containerd://2249881...	prometheus-server	639 mW
containerd://b5853c4f...	grafana	352 mW
containerd://1e777e27...	scaphandre	192 mW

Figure 5-18: Scaphandre dashboard.

OpenShift Consumption Measurement Using Kepler

Production setups are usually based on hardened Kubernetes distributions such as Red Hat's OpenShift Container Platform, which also supports seamless integration with Kepler, as shown in Figure 5-19.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Time Range: Last 30 minutes
Refresh Interval: 30 seconds
Dashboard: API Performance
Apiserver: kube-apiserver
Period: 5m

Kubernetes / Compute Resources / Namespace (Pods)	kubernetes-mixin
Kubernetes / Compute Resources / Namespace (Workloads)	kubernetes-mixin
Kubernetes / Compute Resources / Node (Pods)	kubernetes-mixin
Kubernetes / Compute Resources / Pod	kubernetes-mixin
Kubernetes / Compute Resources / Workload	kubernetes-mixin
Kubernetes / Networking / Cluster	kubernetes-mixin
Kubernetes / Networking / Namespace (Pods)	kubernetes-mixin
Kubernetes / Networking / Pod	kubernetes-mixin
Node Exporter / USE Method / Cluster	node-exporter-mixin
Node Exporter / USE Method / Node	node-exporter-mixin
Power Monitoring / Overview	kepler-mixin
Power Monitoring / Namespace	kepler-mixin
Prometheus / Overview	prometheus-mixin

Figure 5-19: OCP observability dashboard with Kepler integrated.

The "Power Monitoring / Overview" dashboard page shows the system information (e.g., CPU architecture, number of nodes in the cluster), system-level energy consumption, as well as the top 10 energy consuming namespaces. On the other hand, the "Power Monitoring / Namespace" dashboard page shows the namespace- and pod-level power and energy consumptions.

Consumption Based on OpenShift and Keppler

A single node Openshift (SNO) has been set up with Kepler on bare metal and has been evaluated with DPDK l3fwd application, as shown in Figure 5-20. The l3fwd namespace includes 2 pods, each mapped to a NUMA node and NIC. DPDK Pktgen is running on a separate server, able to generate up to 100 Gbps (i.e., 4 x 25 Gbps) to the l3fwd app. A scaled down 24-hour traffic profile, as shown in Figure 5-21, is used as basis for the input traffic, which is load balanced across the 4 links.

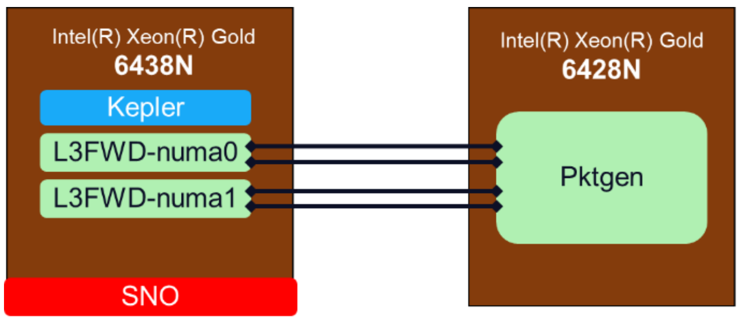


Figure 5-20: DPDK l3fwd app running on single node OpenShift with Kepler integrated.

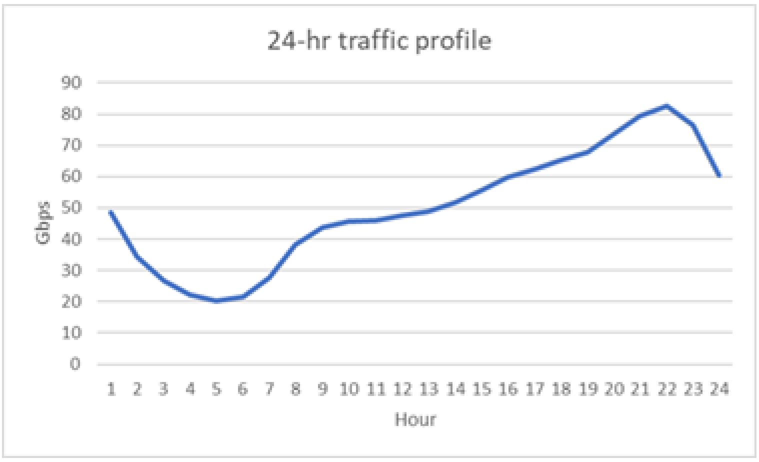
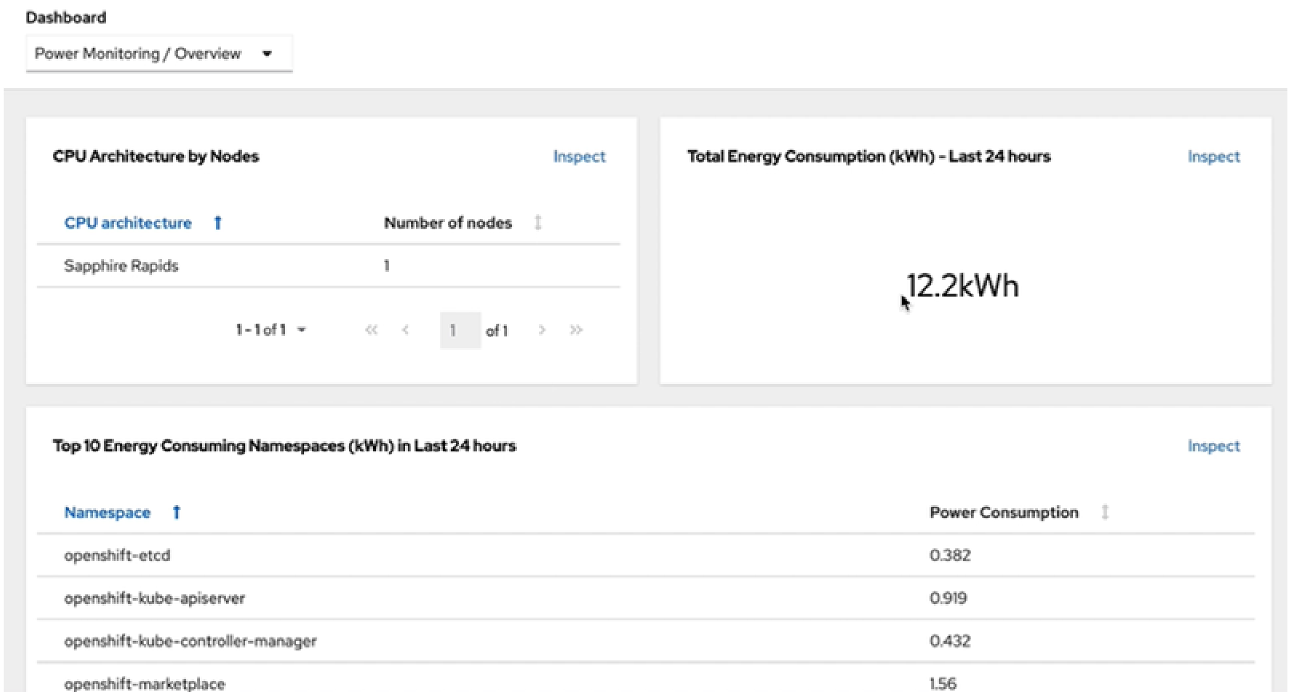


Figure 5-21: Traffic profile generated with Pktgen.

Figure 5-22 and Figure 5-23 show the "Power Monitoring / Overview" and "Power Monitoring / Namespace" dashboard pages, respectively. In the latter, pod level power consumptions for CPU, RAM, GPU (if available) and Other are available in addition to namespace-level power and energy consumptions.

It is important to note that polling applications like DPDK-based applications are observed to always remain in 100% CPU usage despite the actual load. Sleep states are also interrupted by the polling, even when there is no load. In this respect, solutions such as the Intel® Infrastructure Power Manager look into frequency scaling for such cases. The technology has been already used by commercial 5G Core vendors together with DPDK-based UPFs to boost 5G data plane performance, while saving power.


Dashboard
Power Monitoring / Overview

CPU Architecture by Nodes — Inspect

CPU architecture	Number of nodes
Sapphire Rapids	1

1 - 1 of 1 — 1 of 1

Total Energy Consumption (kWh) - Last 24 hours — Inspect

12.2kWh

Top 10 Energy Consuming Namespaces (kWh) in Last 24 hours — Inspect

Namespace	Power Consumption
openshift-etcd	0.382
openshift-kube-apiserver	0.919
openshift-kube-controller-manager	0.432
openshift-marketplace	1.56

Figure 5-22: "Power monitoring / Overview" dashboard page.

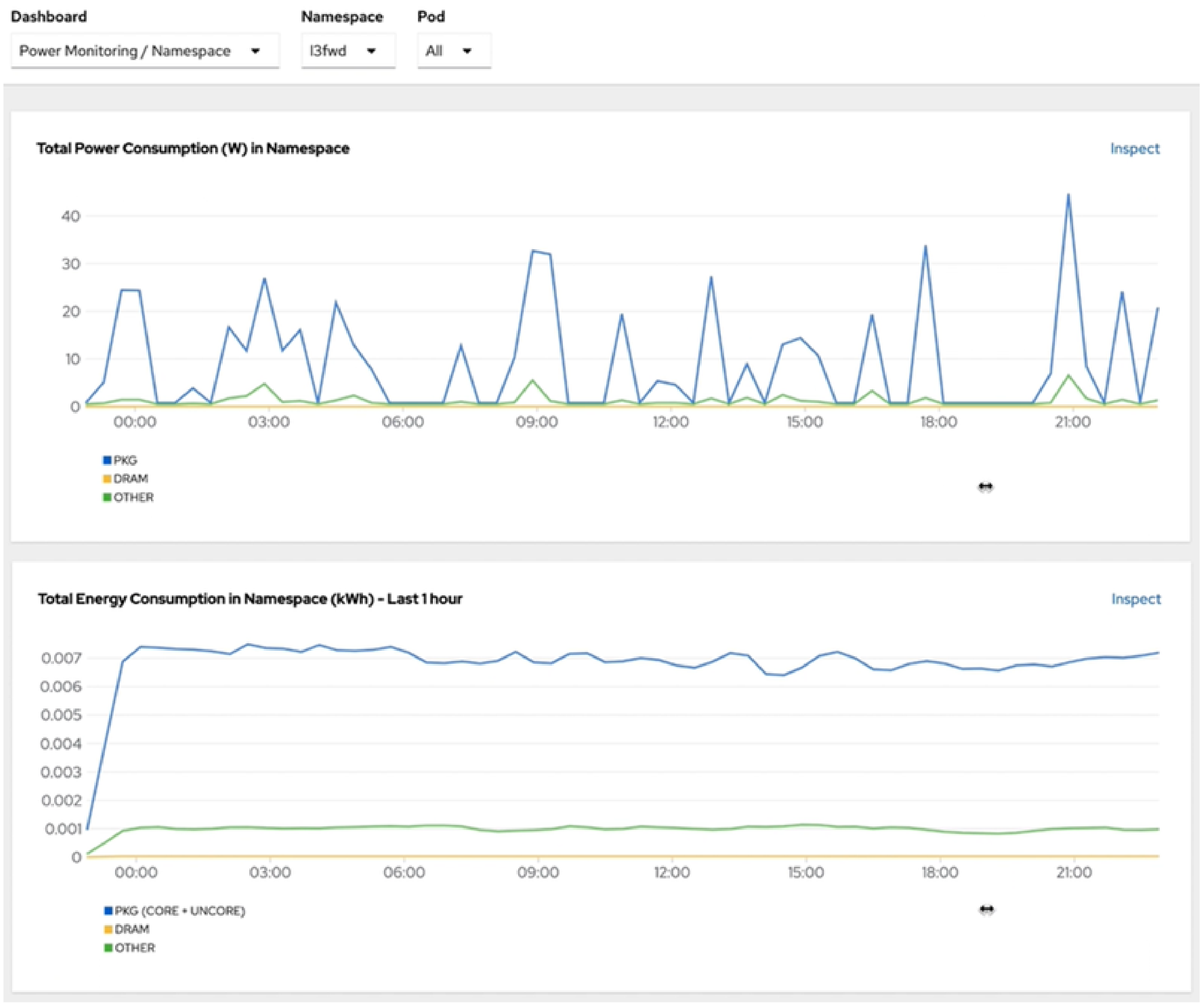


Figure 5-23: "Power monitoring / Namespace" dashboard page.

5.3.2 Measurements Based on the Rating Operator

The rating rules of Rating Operator can be used to model energetic affinities between services and their destination hardware. Furthermore, it empowers users to define transformation rules, enabling them to convert compute resource metrics into KPIs representing energy consumption or carbon footprint. Figure 5-24 illustrates the monitoring of CPU and RAM consumption within a specific namespace of Rating Operator. It also shows the monitoring of carbon footprint at different infrastructure locations or architecture levels, which are transformed from energy consumption using rating rules of Rating Operator.

Align I/O operations to the applied power management schemes and obtain useful energy-aware KPIs to drive energy optimizations

I/O operations are considered from their service counterpart. Each service is then considered as an I/O producer in the context of microservices execution. As telecommunication platforms slowly evolves towards a native support of microservices based execution, this approach is considered both generic and future proof.

The Rating Operator proposes specific rating rules to enable the mapping of resources volumes usage. For each service, several metrics are tracked, including network bandwidth. Within the rules, queries enable the metrics collection, while values are set to enable their transformation towards higher level KPIs (as in metrics x values).

Rating rules enable a mix of metrics to be considered, for example energy metrics obtained from physical devices or logical probes can be related to specific services, or specific resources. The level of precision can be set by the user, either considering a namespace (equivalent of a tenant in the Kubernetes definition) made of services, or a label that can be attached to one or more services.

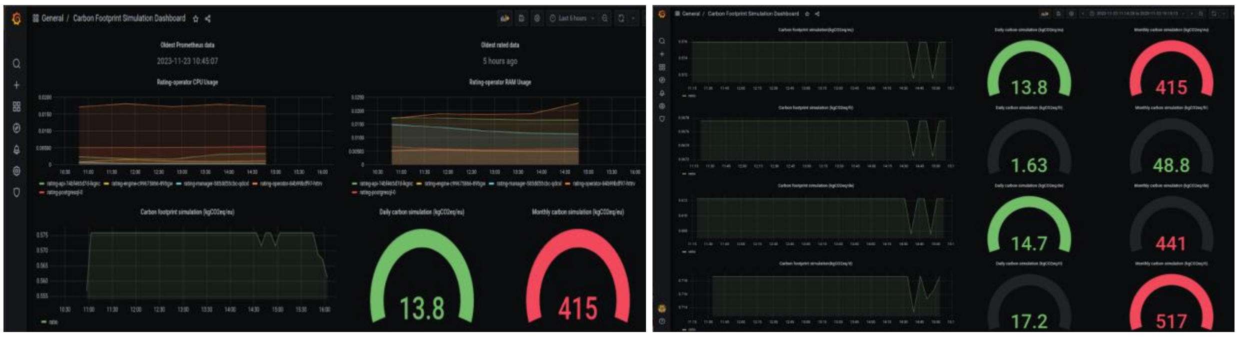

General / Carbon Footprint Simulation Dashboard
Oldest Prometheus data
2023-11-23 10:45:07
Oldest rated data
5 hours ago
Rating-operator CPU Usage
Rating-operator RAM Usage
Carbon footprint simulation (kgCO2eq/au)
Daily carbon simulation (kgCO2eq/au)
Monthly carbon simulation (kgCO2eq/au)
13.8
415
13.8
415
1.63
48.8
14.7
441
17.2
517

Figure 5-24: CPU/RAM usage and carbon footprint monitoring.

5.4 Network Observability and Consumption of Network Equipment

The historical calculation of energy efficiency encompasses the entire network, with all its elements, which includes both legacy cellular technologies and the radio access and core networks, along with data centers (Figure 5-25). It is determined by measuring the amount of electrical energy consumed per unit of transmitted data within a specific time frame, expressed either as Joules per bit or bits per Joule [33], [34].

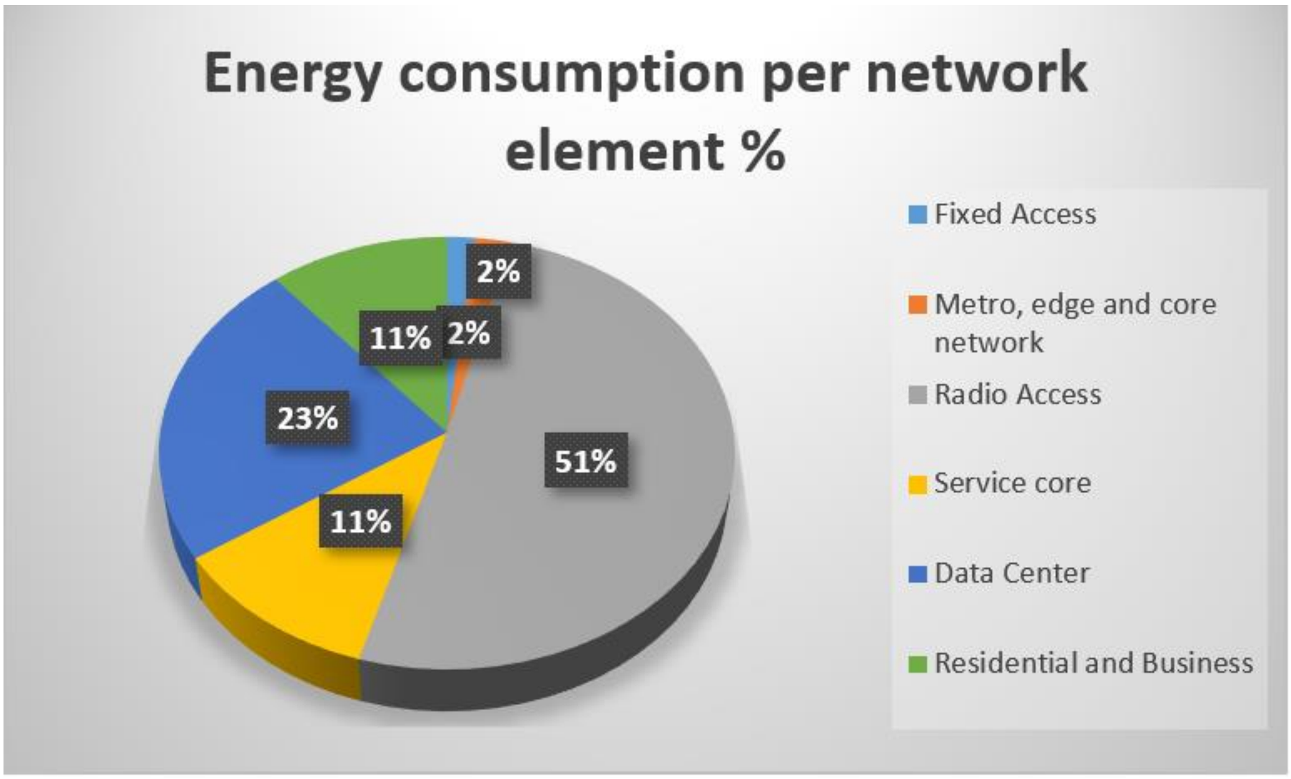


Figure 5-25: Energy consumption breakdown by network element, 2025 [35].

As regards the most expensive segment in terms of energy, the Radio Access one, the network monitoring activities are oriented to allow the four categories of most effective approaches for increasing the energy efficiency[23], nominally [36]:

- Resource allocation: The primary objective is to enhance the energy efficiency by optimally distributing the system's radio resources to minimize power consumption instead of prioritizing throughput. Numerous studies have demonstrated that adopting this approach can result in significant improvements in energy efficiency, albeit with a minor decrease in throughput.
- Network planning and deployment: The second approach involves strategically placing infrastructure nodes to achieve maximum coverage using minimal energy consumption, instead of just optimizing the covered areas. Furthermore, implementing radio devices switch-on/switch-off algorithms and antenna muting techniques allows to further optimize energy usage by adjusting to traffic conditions [37].
- Energy harvesting and transfer: The third method involves harvesting energy from the environment to power communication systems [19].
- Hardware solutions: The aim of this approach is to develop radio communication systems' hardware with a specific focus on energy consumption optimization and implementing significant architectural modifications [38].

In this context, also pushed by the advanced Machine Learning techniques capabilities that are being applied more and more frequently, network monitoring has reached a very fine and granular level (Figure 5-26).

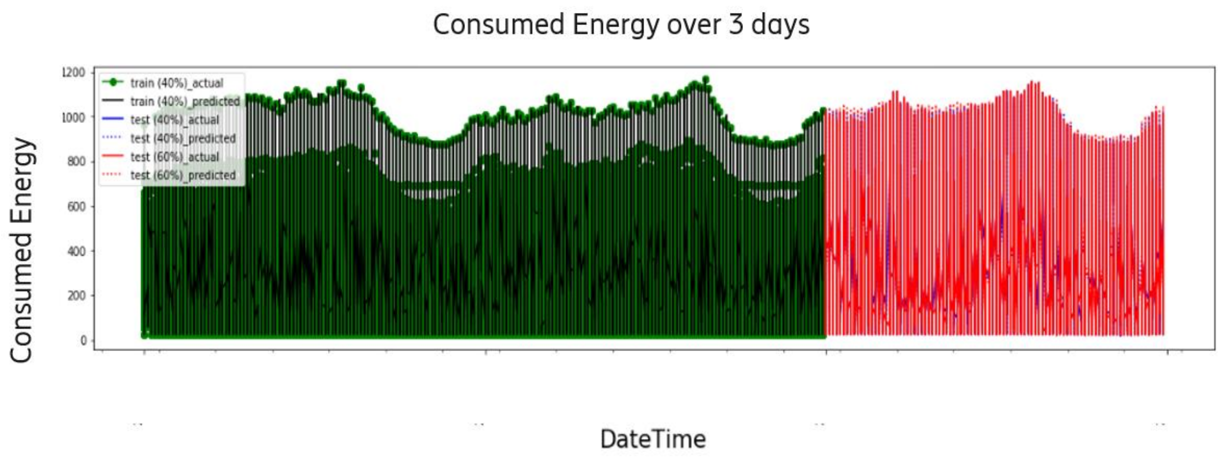


Figure 5-26: ML energy consume prediction.

Among the most important monitored features it is important to mention:

- Radio Resource Control (RRC) Connection Types: Emergency, High Priority Access, Mobile Terminating/Originating Access, Data, Voice Call, and Signalling
- Timing Parameters: MAC SDU data reception timing, transmission intervals, buffer management
- Latency Measurements: Multiple transmission buffer-related latency metrics
- Volume Metrics: Downlink/uplink data radio bearer volumes, signalling radio bearer bits
- Filtered Subclasses: Lower/Higher volume filtering categories

[23] Please note that also if the desire is to achieve energy savings without impacting performances, the technology may provide inherent flexibility to the operators in order to set the best balance between energy efficiency and performances when it is deemed appropriate and justified.

Metro, Edge and Core Network Energy Optimization

Energy saving is approached at three hierarchical levels:

Network Level

- Flexible collaboration between domains and technologies (5G-LTE spectrum optimization)
- Comprehensive intelligent power management
- Data movement minimization through hierarchical caching
- Smart utilization of key 5G features: small cell networks, massive MIMO, device-to-device communications

Site Level

- Renewable energy sources (solar power costs decreased 80% over the last decade)
- Smart lithium batteries
- Cabinet reduction and liquid cooling to minimize air conditioning requirements

Equipment Level

- High-efficient hardware implementation
- Automatic activation/deactivation with shut-down options
- AI/ML and predictive analytics for power efficiency optimization

Power Consumption Models

Specialized literature emphasizes the importance of including precise 5GC deployment software architecture information in network observability, as virtualization technology significantly impacts power consumption patterns.

Refined power models [39] separate contributions from each active domain:

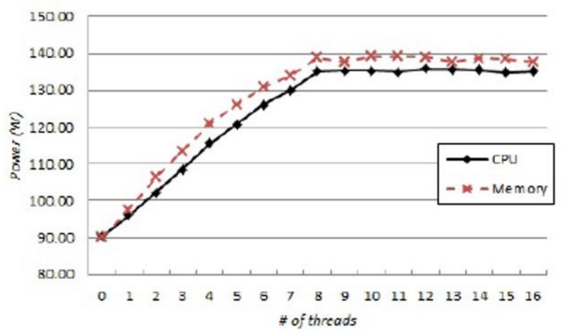
Total Power Model:

$$P_{total} = P_{baseline} + \sum_{K=1}^{N} P_{domain}(k)$$

Where $P_{domain}(k)$ is the power consumed by an active domain k, and N is the number of domains, and each domain factor can be referred to a multi-dimensional linear weighted power model:

$$P_{domain} = c_0 + c_1 P_{CPU} + c_2 P_{cache} + c_3 P_{DRAM} + c_4 P_{disk} + \cdots$$

The different contributions are characterized starting from careful testing campaigns (e.g., Figure 5-27 [39]) and further indicate the required quantities that need to be constantly observed and collected.



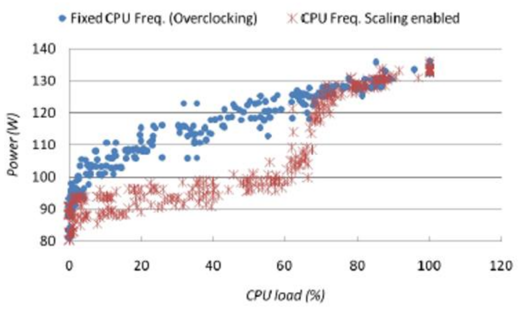


Figure 5-27: Power vs threads, CPU frequency and Memory [39].

5.4.1 Power Consumption Monitoring on Cloud-Native 5GS

To validate power consumption monitoring concepts in cloud-native 5G environments, we built a proof of concept in 5G/6G testbed (Figure 5-28). We extended cloud-native 5G systems, featuring zero-touch provisioning and end-to-end slicing, with a power measurement toolset. We deployed Netio for power outlet level measurements, embedded RRU and IaaS tools for RRU input/output and CPU power metering, and the Scaphandre tool for process-level metering to measure the consumption of BBU, 5G CN, and application components. The qMON test automation tool was utilized to control and predict user traffic patterns and to evaluate cloud-native system power consumption under real traffic load.

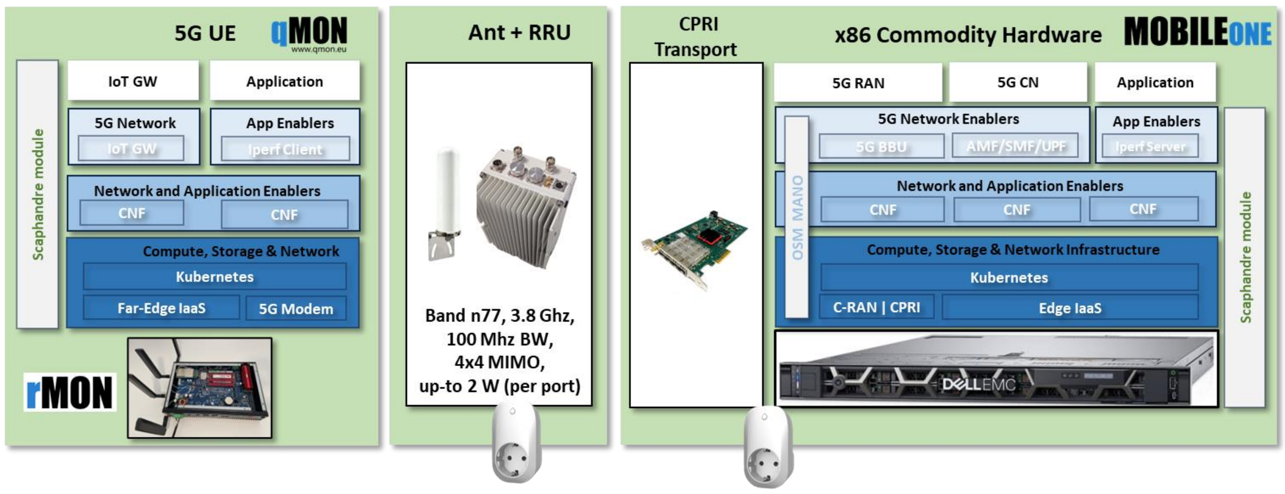
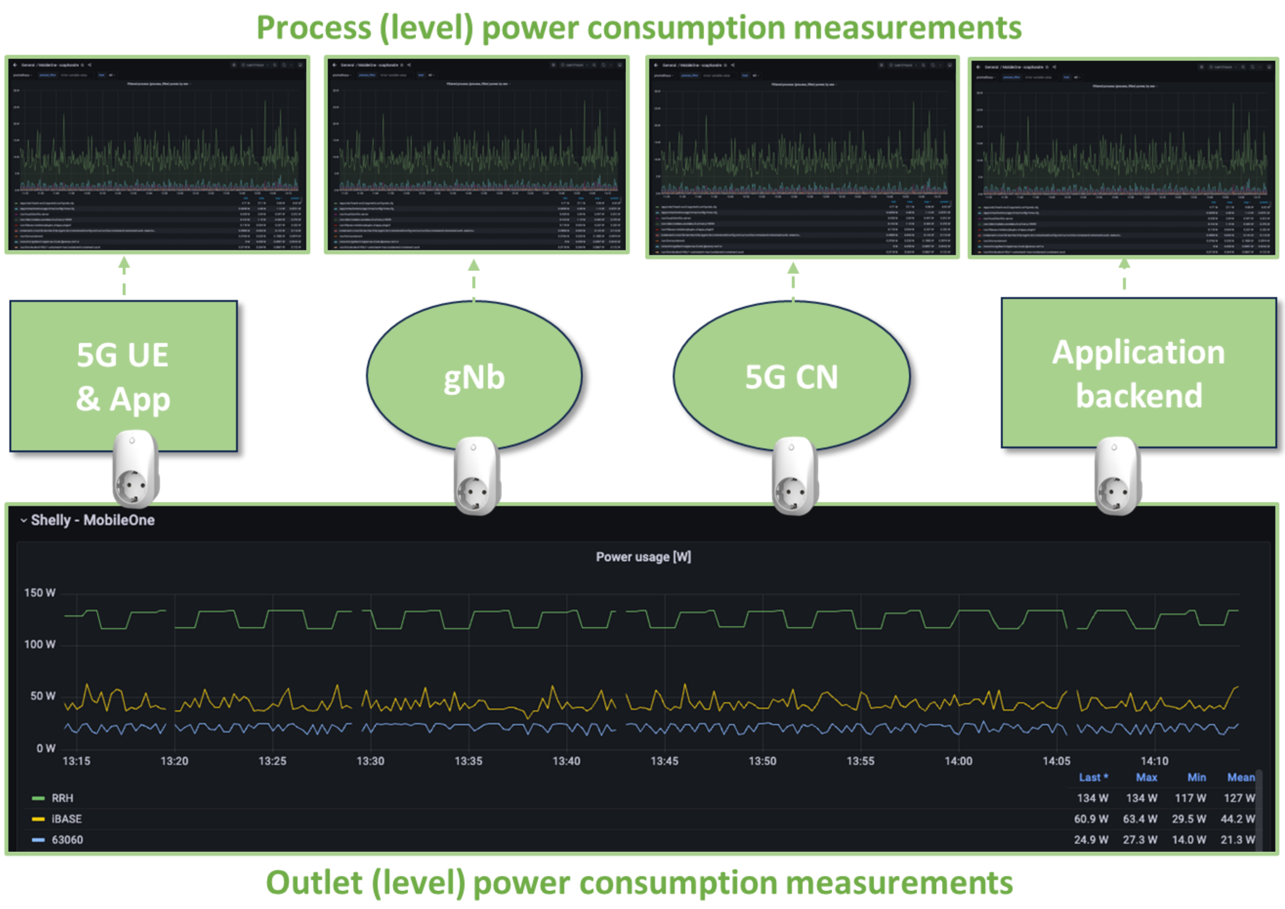


Figure 5-28: 5G/6G Pilot Environment.

The deployed power measurement toolset enables active measurement of the 5G system's power consumption from an end-to-end perspective. This includes 5G UE, RRU, BBU, 5G CN, and application components, providing full visibility of all factors — both hardware and software — that impact the total power consumption of the mobile environment (Figure 5-29).



	Last *	Max	Min	Mean
RRH	134 W	134 W	117 W	127 W
iBASE	60.9 W	63.4 W	29.5 W	44.2 W
63060	24.9 W	27.3 W	14.0 W	21.3 W

Figure 5-29: Measuring power consumption taking into consideration end-to-end perspective.

To verify the proposed power metering approach, we have deployed 5G system on an x86 NFVI environment, which allows us to run 5G NR (virtual BBU) and 5G core network functions on a single Kubernetes instance. This supports container-based deployment of network functions and MANO-compliant orchestration. We deployed 5GCN (virtual core network) and 5G NR (virtual BBU) as network functions (NF) and corresponding network services (NS) using Kubernetes deployment principles. With virtual network function descriptors (VNFD) and network service descriptors (NSD), we can easily reconfigure key 5G NR and Core network parameters, such as used RRU band and bandwidth, power per radio port on RRU, MIMO Level, TDD mode, slicing configuration, user bandwidth profile, and user traffic patterns.

We have conducted a series of tests to verify the prepared environment. In the first test, we aimed to observe the difference between an idle and an active user — where the user is not generating any traffic (UE in idle mode) and RAN is also in idle mode or when the user is fully utilizing available 5G RAN resources (UE in active mode). We prepared a test methodology where we generated TCP-based DL traffic for a duration of 2 minutes, followed by an idle time of 1 minute. This testing was repeated in cycles. Figure 5-30 provides an indicative view on relations and dynamics of power consumption at different components while testing with before described (cyclic) network load. Based on the observed results, we can conclude:

- During the active test cycle, when the 5G UE is generating traffic, power consumption increases significantly.
- Even when the user is idle, the 5G system and 5G UE consume significant power. This includes power usage in hardware standby mode (RRU, IaaS server), 5G NR and 5GCN standby mode (BBU, AMF, SMF), and application standby mode.
- If user traffic follows a deterministic pattern, then power consumption (RRU, 5G CN, Iperf Server) also exhibits a deterministic pattern.

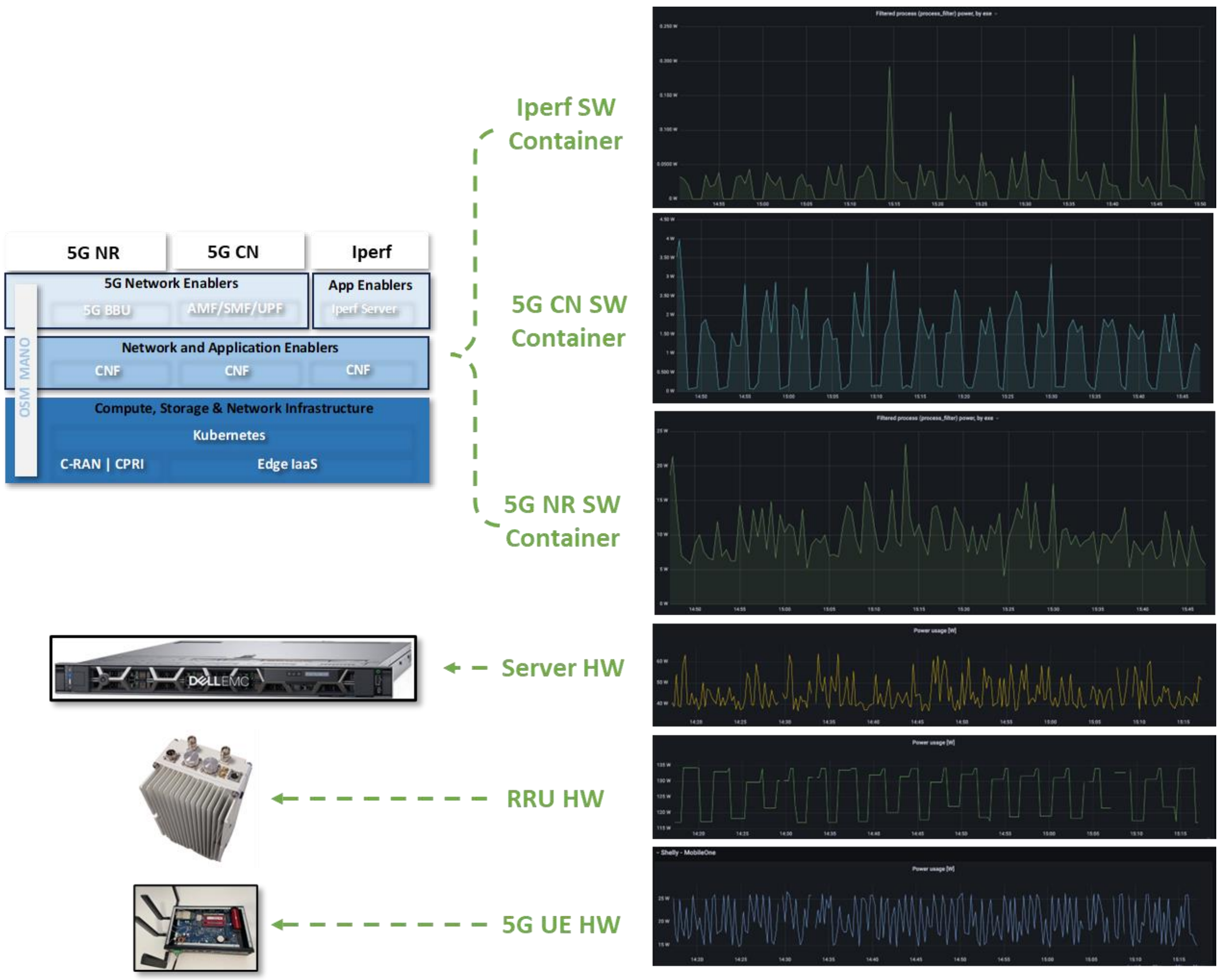


Figure 5-30: Measuring the power consumption of the 5G system, breaking down the hardware and software components (and indicative view on relations and dynamics of power consumption at different components while testing with cyclic network load).

6 Adoption of Enabling Technologies in the 6Green Service-based Architecture

In the current section, we shortly refer to the adoption of the aforementioned technologies towards the development of the 6Green Service-based Architecture (SBA), as shown in Figure 6-1.

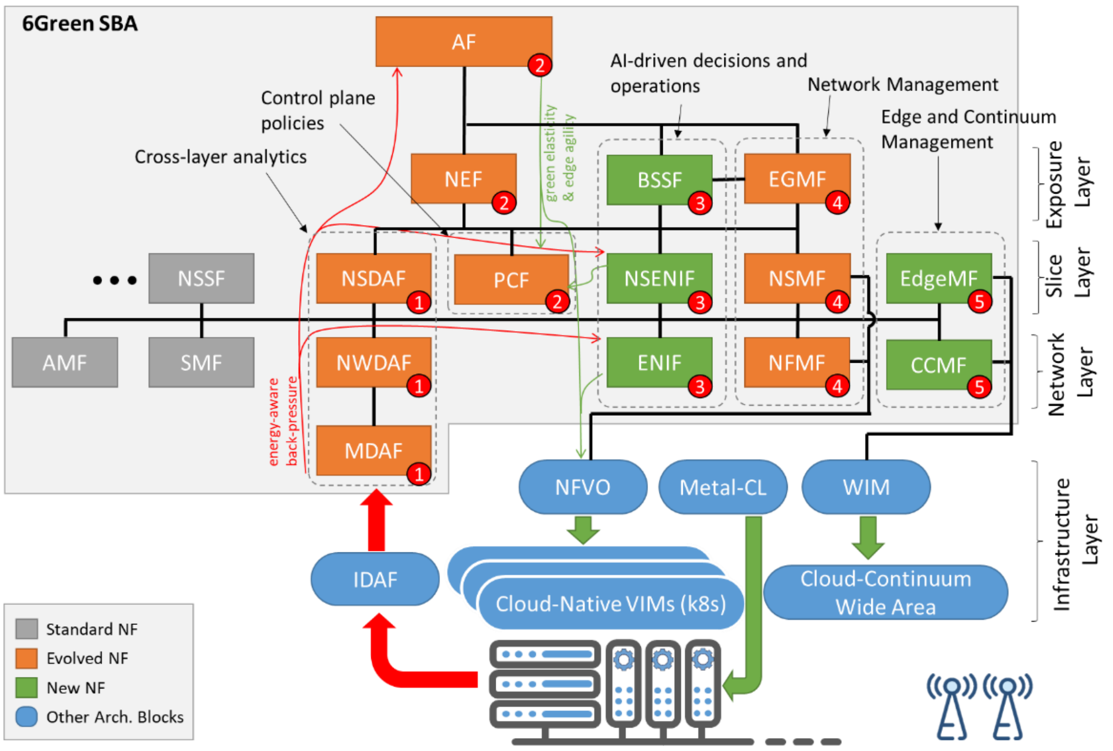


Figure 6-1: The 6Green SBA framework.

Following, in Table 11 we highlight the set of main technologies that are adopted towards the development of each of the components of the 6Green SBA, while a short description for such an adoption is provided in Table 12.

Table 11: Mapping among the Enabling technologies and the components of the SBA.

Enabling Technology	MDAF	NWDAF	NSDAF	PCF/ NSPCF	NEF	AF	BSSF	ENIF/ NSENIF	EGMF/ NSMF/NF	EdgeMF	CCMF
Traffic Offloading					X	X					
Connectivity (Wide-Area Infrastructure Manager)								X			X
Infrastructure as a Code (MetalCL)								X		X	X
ZeroOps and Automation in Infra-structure Management (NFVCL)							X	X			

Enabling Technology	MDAF	NWDAF	NSDAF	PCF/ NSPCF	NEF	AF	BSSF	ENIF/ NSENIF	EGMF/ NSMF/NF	EdgeMF	CCMF
FaaS programming model								X			
RAN power management			X					X			
Optimal deployment of network slices			X					X	X		
Data Fusion and Profiling	X	X	X				X	X			
Network observability		X	X		X		X	X			
Dynamic QoS management		X		X	X				X		

Table 12: Short description for the adoption of Enabling technologies per component of the SBA.

SBA Component	Main usage of enabling technologies
MDAF	The main enabling technology adopted by the MDAF is the Data Fusion and Profiling one which is exploited to generate more complex data (e.g., forecasting the power consumption of cloud native components).
NWDAF	The NWDAF adopts two main enabling technologies: Data Fusion and Profiling and Network Observability. The former is exploited to generate more complex data by fusing the ones produced by the MDAF and those produced by the other NFs (e.g., AMF, SMF, etc.). The latter is a crucial part of the NWDAF since this is the NF devoted to spreading observability information from the sources (i.e., MDAF, AMF, SMF, etc.) to all the other NFs that request them.
NSDAF	The NSDAF enables the network slice analysis in the context of the 6G architectures, for optimal energy resources consumption, applied in the cloud native environment. The envisioned SBA integration of the NSDAF component together with NWDAF (seen as functional sub-component of the NF) will interact with the other NFs for green infrastructure and services implementation, with support of 6Green orchestration and automation tools.
PCF/ NSPCF	PCF serves as a crucial enabling technology that manages network policies and enforces rules for data traffic and Quality of Service (QoS). It dynamically adjusts policies based on real-time network conditions and subscriber data, ensuring efficient resource utilization and optimal service delivery.
NEF	NEF facilitates secure and efficient exposure of network services and capabilities to third-party applications (AF). It provides a standardized interface for external applications to interact with the network, ensuring proper access control, policy enforcement and enabling the traffic offloading mechanism.

SBA Component	Main usage of enabling technologies
AF	AF functionalities are provided by the Vertical Application Orchestrator which is used for managing the deployment and real-time operations of vertical applications, interfacing seamlessly with end users and network orchestration systems. Thus, it decouples application layer management from network layer management, providing an interface for users to manage applications and their features, and handling the entire lifecycle of vertical applications. This includes requesting cloud resources, configuring network slices, and implementing Green Elasticity and Edge Agility based on workload demands. It also provides capabilities to trigger network and slice configuration changes through Service-Based Interface (SBIs) with the BSSF, dynamically configuring the network to meet specific application needs in real-time. It operates infrastructure-agnostically, utilizing a dynamic intent-based system for real-time intent negotiation and reconfiguration, optimizing resource usage and network performance with Energy-aware Backpressure information flows.
BSSF	Intent management facilitates a ZeroOps and automated network management by abstracting underlying configurations from verticals. The BSSF acts as an aggregator of slice requests, enabling the upper layers to automate processes more reliably. It also allows for profiling in intent management based on policies assigned to specific vertical types. As a consequence of requesting a slice, network monitoring is also present in BSSF.
ENIF / NSENIF	One of the main enabling technologies adopted in ENIF/NSENIF regard the mechanisms that support optimal deployment and lifecycle management of a network slice, considering resources in the RAN, transport and core network part, as well as deployment in serverless and non-serverless environments. Intent lifecycle management is supported from the specification of the intent towards its validation and its monitoring during the lifetime of a service deployment and operation. Dynamic policies management is applied to satisfy the requested intent. Data fusion and profiling mechanisms are used to continuously monitor various performance metrics over the infrastructure, analyzed data and proceed to decision making. ZeroOps, automation and infrastructure as a code principles are exploited to increase automation and distributed intelligence of the provided services by ENIF/NSENIF.
EGMF / NSMF / NFMF	NSMF deploys an end-to-end NSI for each network connectivity demand expressed by AFs. To do this, NSMF consumes the management services of other 6Green NFs using SBA. For example, NSMF consumes NFMF services to configure the deployed VNFs/CNFs by NFVO to establish a new NSI or alter an existing one. To protect the management services from unauthorized AFs, EGMF, another 6Green NF, is responsible for securely exposing the management services. Finally, whenever a green decision regarding modifying an NSI is made, NSMF, as an actuator, is in charge of applying that decision in the 6Green SBA.

SBA Component	Main usage of enabling technologies
EdgeMF	EdgeMF allows the SBA to provide computing resources at the edge to vertical applications, enabling Edge Agility and Green Elasticity mechanisms. It achieves this by managing Edge Data Networks (EDN) objects that provide data and compute services in a specific edge zone, as well as by managing the compute resources that are made up of one or more EDNs. For this purpose, EdgeMF will leverage the services of the VAO and the NSMF, and will require services from the CCMF and the MetalCL infrastructure component.
CCMF	The CCMF enables the SBA to maintain a repository of computing resources to be used to deploy vertical applications, e.g., from Kubernetes clusters.

7 Conclusions

In accordance with the objectives of WP2, in this deliverable we have listed a set of enabling technologies that are developed in the 6Green Project and are adopted for the development of the 6Green Service-based Architecture (SBA).

A wide range of enabling technologies are detailed, including network connectivity management and traffic offloading mechanisms; cloud-native orchestration mechanisms considering approaches that take advantage of service-mesh techniques, as well as automation mechanisms based on infrastructure as a code, ZeroOps and continuous automation principles; power management mechanisms for the core, transport and access part of the continuum by considering serverless workloads; network slice lifecycle management and optimization techniques, including energy-aware network slice management in O-RAN and multi-provider settings; and green observability and profiling mechanisms.

Upon the description of the development of the set of enabling technologies, a mapping among the detailed technologies and the components of the 6Green SBA is provided. This mapping is aligned with the development of software prototypes for the enabling technologies as detailed in D2.4, as well as with the development of the 6Green SBA, as detailed in D3.3 and D3.4.

References

[1] 3GPP, “5G Network Resource Model (NRM)”, 3GPP TS 28541 V17111, 2024.

[2] A. Farrel et al., “A Framework for Network Slices in Networks Built from IETF Technologies”, n. 9543. en Request for Comments, [Online]. Available at: https://www.rfc-editor.org/info/rfc9543

[3] B. Wu, D. Dhody, R. Rokui, T. Saad, y J. Mullooly, “A YANG Data Model for the RFC 9543 Network Slice Service”, Internet Engineering Task Force, Internet-Draft draft-ietf-teas-ietf-network-slice-nbi-yang-10, Mar. 2024, [Online].
Available at: https://datatracker.ietf.org/doc/draft-ietf-teas-ietf-network-slice-nbi-yang/10/

[4] «TeraFlowSDN | TeraFlow». Available at: https://www.tfs.etsi.org//

[5] 6Green D4.1, https://www.6green.eu/

[6] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271

[7] D. de la Osa Mostazo, P. Armingol Robles, Ó. G. de Dios, and J. Pedro Fernández-Palacios Giménez, ‘Lessons learned from IP routers power measurements and characterization’, in *2024 15th International Conference on Network of the Future (NoF)*, Oct. 2024, pp. 245–253. doi: 10.1109/NoF62948.2024.10741444.

[8] R. Bolla, R. Bruschi, A. Gallo, C. Lombardo and N. S. Martinelli, "To Scale or Not to Scale? Understand the Overhead of Container Scaling Operations," 2025 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT), Lucca, Italy, 2025, pp. 890-894, doi: 10.1109/DCOSS-IoT65416.2025.00135. https://ieeexplore.ieee.org/document/11096301

[9] Intel, “Preboot Execution Environment (PXE) Specification”, V2.1, 1999, [Online].
Available: https://web.archive.org/web/20131102003141/http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf

[10] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, “The State of Serverless Applications: Collection, Characterization, and Community Consensus”, IEEE Transactions on Software Engineering, 2021.

[11] S. Risco, C. Alarcón S. Langarita, M. Caballer, and G. Moltó, “Rescheduling serverless workloads across the cloud-to-edge continuum”, Future Generation Computer Systems, vol. 153, pp. 457–466, Apr. 2024.

[12] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, “Wisefuse’: Workload Characterization and DAG Transformation for Serverless Workflows”, in Proceedings of the ACM on Measurement and Analysis of Computing Systems, vol. 6. Association for Computing Machinery, 2022, pp. 1–28, issue: 2.

[13] Z. Xu, L. Zhou, W. Liang, Q. Xia, W. Xu, W. Ren, H. Ren, and P. Zhou, “Stateful Serverless Application Placement in MEC with Function and State Dependencies”, IEEE Transactions on Computers, pp. 1-14, 2023.

[14] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. Richard Yu, and T. Huang, “When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues”, IEEE Wireless Communications, pp. 1–8, 2021.

[15] C. Cicconetti, M. Conti, and A. Passarella, “FaaS execution models for edge applications”, Pervasive and Mobile Computing, vol. 86, 2022.

[16] D. Schafer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, “Tasklets: ‘Better than Best-Effort’ Computing”, in 2016 25th International Conference on Computer Communication and Networks (ICCCN). Waikoloa, HI, USA: IEEE, Aug. 2016, pp. 1-11.

[17] C. Cicconetti, M. Conti, and A. Passarella, “A Decentralized Framework for Serverless Edge Computing in the Internet of Things”, IEEE Trans. on Network and Service Management, pp. 1–1, 2020.

[18] H. Tian, Y. Zheng, and W.Wang, “Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud”, ACM SoCC, pp. 139-151, 2019.

[19] Ulukus, S.; Yener, A.; Erkip, E.; Simeone, O.; Zorzi, M.; Grover, P.; Huang, K. “Energy harvesting wireless communications: A review of recent advances”, IEEE J. Sel. Areas Commun. 2015, 33, 360-380, [Online]. Available: https://ieeexplore.ieee.org/document/7010878

[20] Esmaeil Amiri, Ning Wang, Mohammad Shojafar, Mutasem Q. Hamdan, Chuan Heng Foh, and Rahim Tafazolli, “Deep Reinforcement Learning for Robust VNF Reconfigurations in O-RAN”, IEEE Trans. on Netw. and Serv. Manag. 21, 1 (Feb. 2024), 1115-1128.

[21] N. Sen and A. F. A, “Towards Energy Efficient Functional Split and Baseband Function Placement for 5G RAN”, 2023 IEEE 9th International Conference on Network Softwarization (NetSoft), Madrid, Spain, 2023, pp. 237-241.

[22] Y. Shi, Y. E. Sagduyu and T. Erpek, “Reinforcement Learning for Dynamic Resource Optimization in 5G Radio Access Network Slicing”, 2020 IEEE 25th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Pisa, Italy, 2020, pp. 1-6.

[23] N. Sen and A. F. A, “Intelligent Admission and Placement of O-RAN Slices Using Deep Reinforcement Learning”, 2022 IEEE 8th International Conference on Network Softwarization (NetSoft), Milan, Italy, 2022, pp. 307-311.

[24] N. Fryganiotis, E. Stai, I. Dimolitsas, A. Zafeiropoulos, S. Papavassiliou “Dynamic, Reconfigurable and Green Network Slice Admission Control and Resource Allocation in the O-RAN Using Model Predictive Control”, IFIP Networking, Thessaloniki, Greece, June 2024.

[25] A. Varasteh, B. Madiwalar, A. Van Bemten, W. Kellerer and C. Mas-Machuca, “Holu: Power-Aware and Delay-Constrained VNF Placement and Chaining”, in IEEE Transactions on Network and Service Management, vol. 18, no. 2, pp. 1524-1539, June 2021.

[26] Watters, Lawrence J. “Reduction of Integer Polynomial Programming Problems to Zero-One Linear Programming Problems”, Operations Research 15, no. 6 (1967):1171-74, [Online]. Available: http://www.jstor.org/stable/168623.

[27] A. Zafeiropoulos, N. Fryganiotis, P. Maratos, C. Vassilakis, E. Stai, and S. Papavassiliou, “Leveraging knowledge graphs for intent lifecycle management in the computing continuum,” in 2025 IEEE Global Communications Conference, 2025.

[28] Tzanettis I, Androna C-M, Zafeiropoulos A, Fotopoulou E, Papavassiliou S., “Data Fusion of Observability Signals for Assisting Orchestration of Distributed Applications”, Sensors, 2022, 22(5):2061, [Online]. Available: https://doi.org/10.3390/s22052061

[29] Etienne-Victor Depasquale, Franco Davoli and Humaira Rajput “Dynamics of Research into Modeling the Power Consumption of Virtual Entities Used in the Telco Cloud”, Sensors, 2023, 23(1):255, [Online]. Available: https://doi.org/10.3390/s23010255

[30] Kuranage, Menuka; Hanser, Elisabeth; Nuaymi, Loutfi; Bouabdallah, Ahmed; Bertin, Philippe; Al-Dulaimi, Anwer, "AI-assisted proactive scaling solution for CNFs deployed in Kubernetes", 265-273. doi: 10.1109/CloudNet59005.2023.10490067.

[31] Spatharakis, Dimitrios, et al. "Distributed resource autoscaling in kubernetes edge clusters", 2022 18th International Conference on Network and Service Management (CNSM). IEEE, 2022.

[32] A. Zafeiropoulos et al., "Benchmarking and Profiling 5G Verticals' Applications: An Industrial IoT Use Cas", 2020 6th IEEE Conference on Network Softwarization (NetSoft), Ghent, Belgium, 2020, pp. 310-318, doi: 10.1109/NetSoft48620.2020.9165393.

[33] Humar, I.; Ge, X.; Xiang, L.; Jo, M.; Chen, M.; Zhang, J. "Rethinking energy efficiency models of cellular networks with embodied energy", IEEE Netw. 2011, 25, 40-49 [Online]. Available: https://ieeexplore.ieee.org/document/5730527

[34] Andrews, J.G.; Buzzi, S.; Choi, W.; Hanly, S.; Lozano, A.; Soong, A.C.K.; Zhang, J.C. "What Will 5G Be?", IEEE J. Sel. Areas Commun. 2014, 32, 1065-1082. [Online]. Available: https://ieeexplore.ieee.org/document/6824752

[35] Ioannis P. Chochliouros, Michail-Alexandros Kourtis, Anastasia S. Spiliopoulou, Pavlos Lazaridis, Zaharias Zaharis, Charilaos Zarakovitis and Anastasios Kourtis, "Energy Efficiency Concerns and Trends in Future 5G Network Infrastructures", Energies, 2021, 14(17):5392, [Online]. Available: https://doi.org/10.3390/en14175392

[36] Zappone, A.; Jorswieck, E. "Energy efficiency in wireless networks via fractional programming theory", Found. Trends Commun. Inf. Theory 2015, 11, 185-396, [Online]. Available: https://www.nowpublishers.com/article/Details/CIT-088

[37] Oh, E.; Son, K.; Krishnamachari, B. "Dynamic base station switching-on/off strategies for green cellular networks", IEEE Trans. Wirel. Commun. 2013, 12, 2126-2136. [Online]. Available: https://ieeexplore.ieee.org/document/6489498

[38] Han, C.; Harrold, T.; Armour, S.; Krikidis, I.; Videv, S.; Grant, P.M.; Haas, H.; Thompson, J.S.; Ku, I.; Wang, C.X.; et al. "Green radio: Radio techniques to enable energy-efficient wireless networks", IEEE Commun. May 2011, 49, 46-54, [Online]. Available: https://ieeexplore.ieee.org/document/5783984

[39] Qingwen Chen, Paola Grosso, Karel van der Veldt, Cees de Laat, Rutger Hofman, Henri Bal, "Profiling energy consumption of VMs for green cloud computing", 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing, Sydney, NSW, Australia, 2-14 December 2011, IEEE, doi: 10.1109/DASC.2011.131

Annex A: Blueprint Deployment and Lifecycle Management Workflows

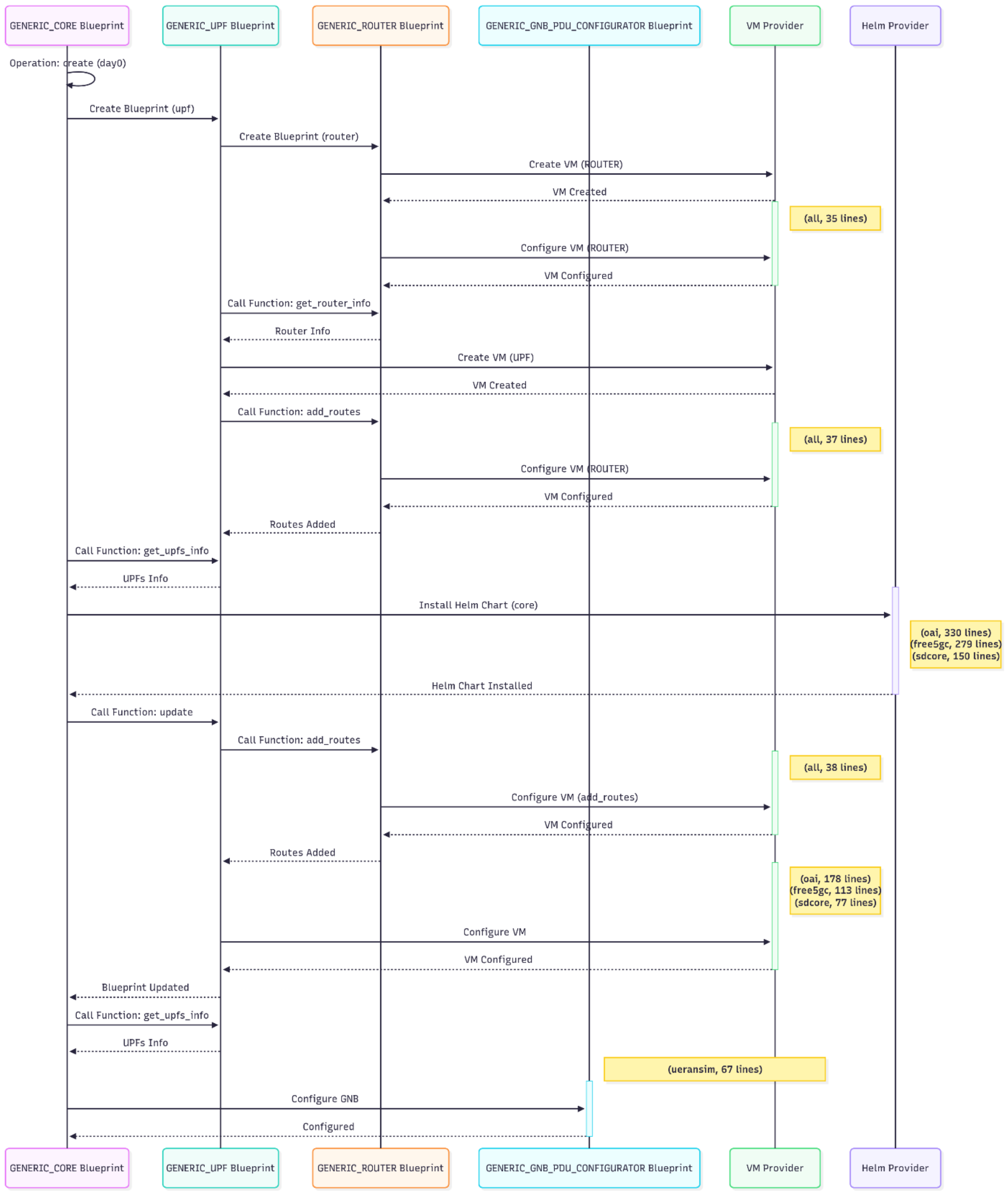


Figure A-1: Operations required to create a core with the UPF provisioned in a VM.

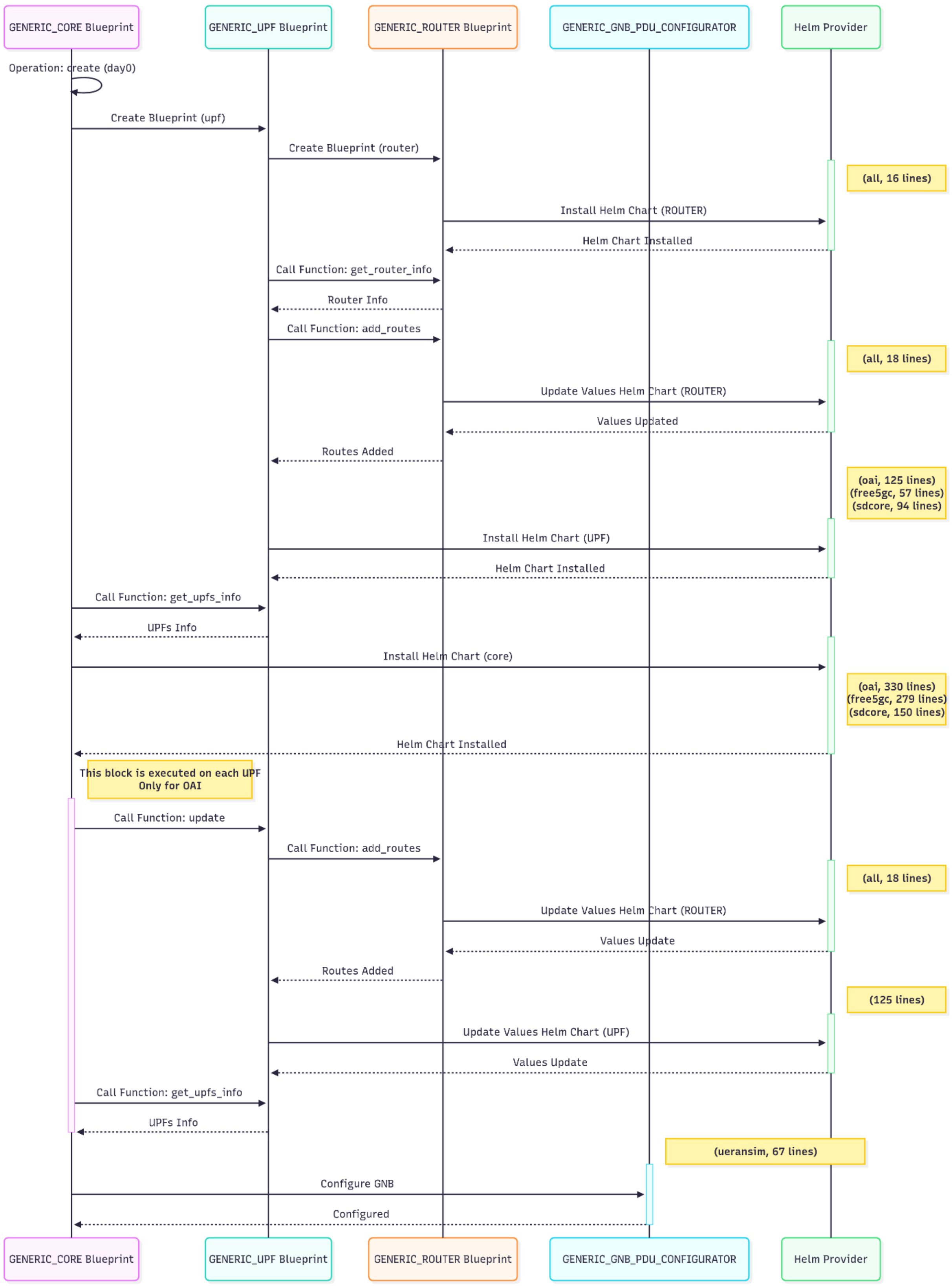

GENERIC_CORE Blueprint
GENERIC_UPF Blueprint
GENERIC_ROUTER Blueprint
GENERIC_GNB_PDU_CONFIGURATOR
Helm Provider
Operation: create (day0)
Create Blueprint (upf)
Create Blueprint (router)
(all, 16 lines)
Install Helm Chart (ROUTER)
Helm Chart Installed
Call Function: get_router_info
Router Info
Call Function: add_routes
(all, 18 lines)
Update Values Helm Chart (ROUTER)
Values Updated
Routes Added
(oai, 125 lines) (free5gc, 57 lines) (sdcore, 94 lines)
Install Helm Chart (UPF)
Helm Chart Installed
Call Function: get_upfs_info
UPFs Info
Install Helm Chart (core)
(oai, 330 lines) (free5gc, 279 lines) (sdcore, 150 lines)
Helm Chart Installed
This block is executed on each UPF Only for OAI
Call Function: update
Call Function: add_routes
(all, 18 lines)
Update Values Helm Chart (ROUTER)
Values Update
Routes Added
(125 lines)
Update Values Helm Chart (UPF)
Values Update
Call Function: get_upfs_info
UPFs Info
(ueransim, 67 lines)
Configure GNB
Configured

Figure A-2: Operations required to create a core with the UPF provisioned in a pod.

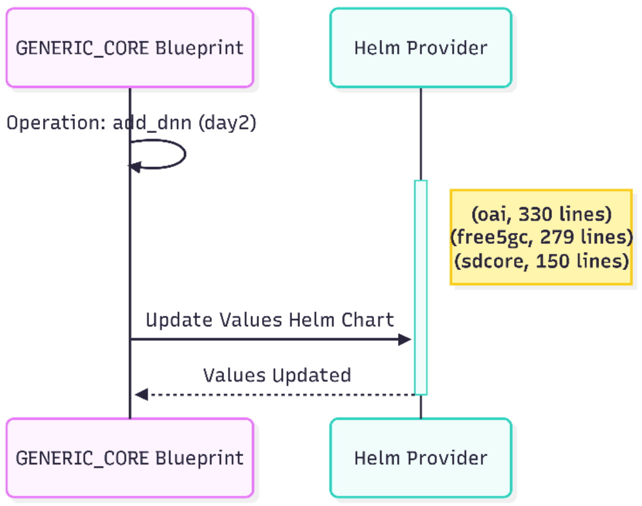


Figure A-3: Operations required to add a DNN.

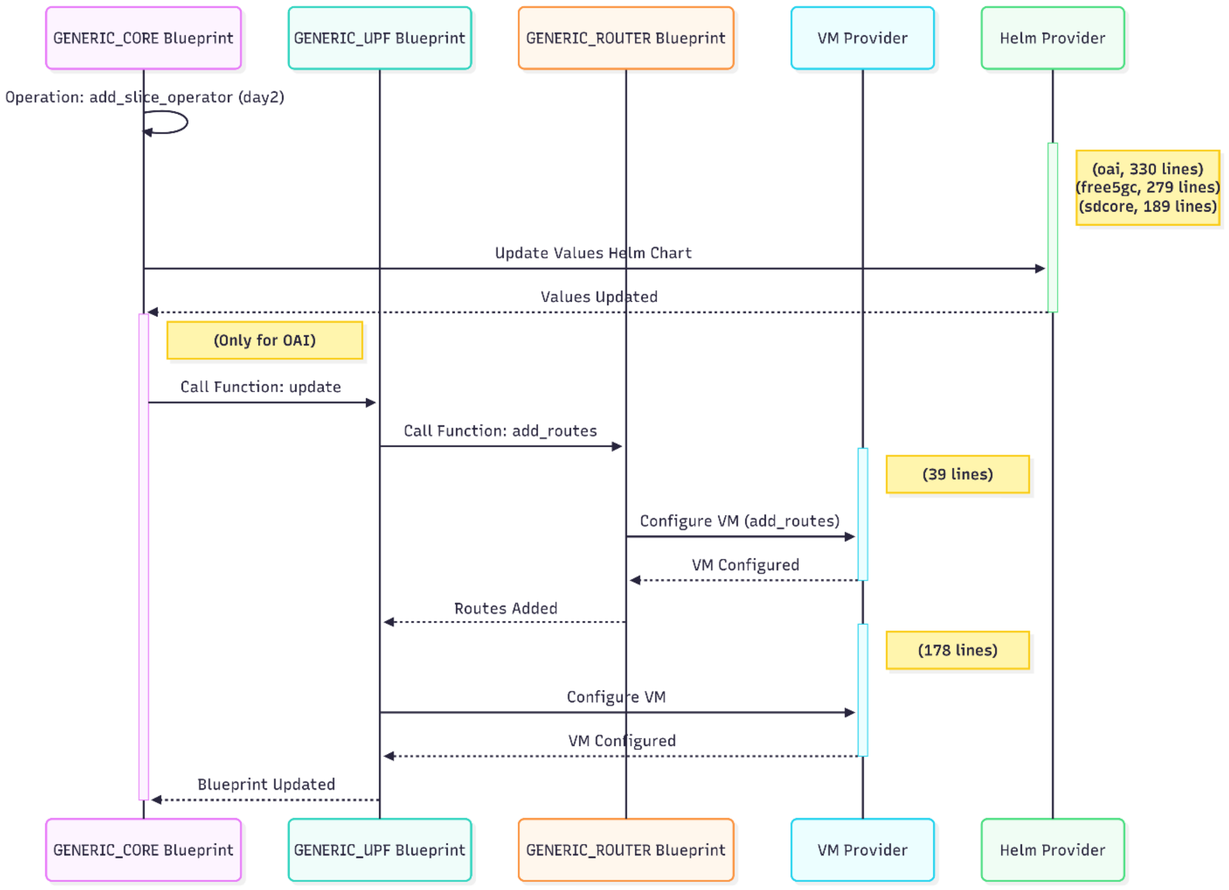


Figure A-4: Operations required to add a slice with the UPF provisioned in a VM.

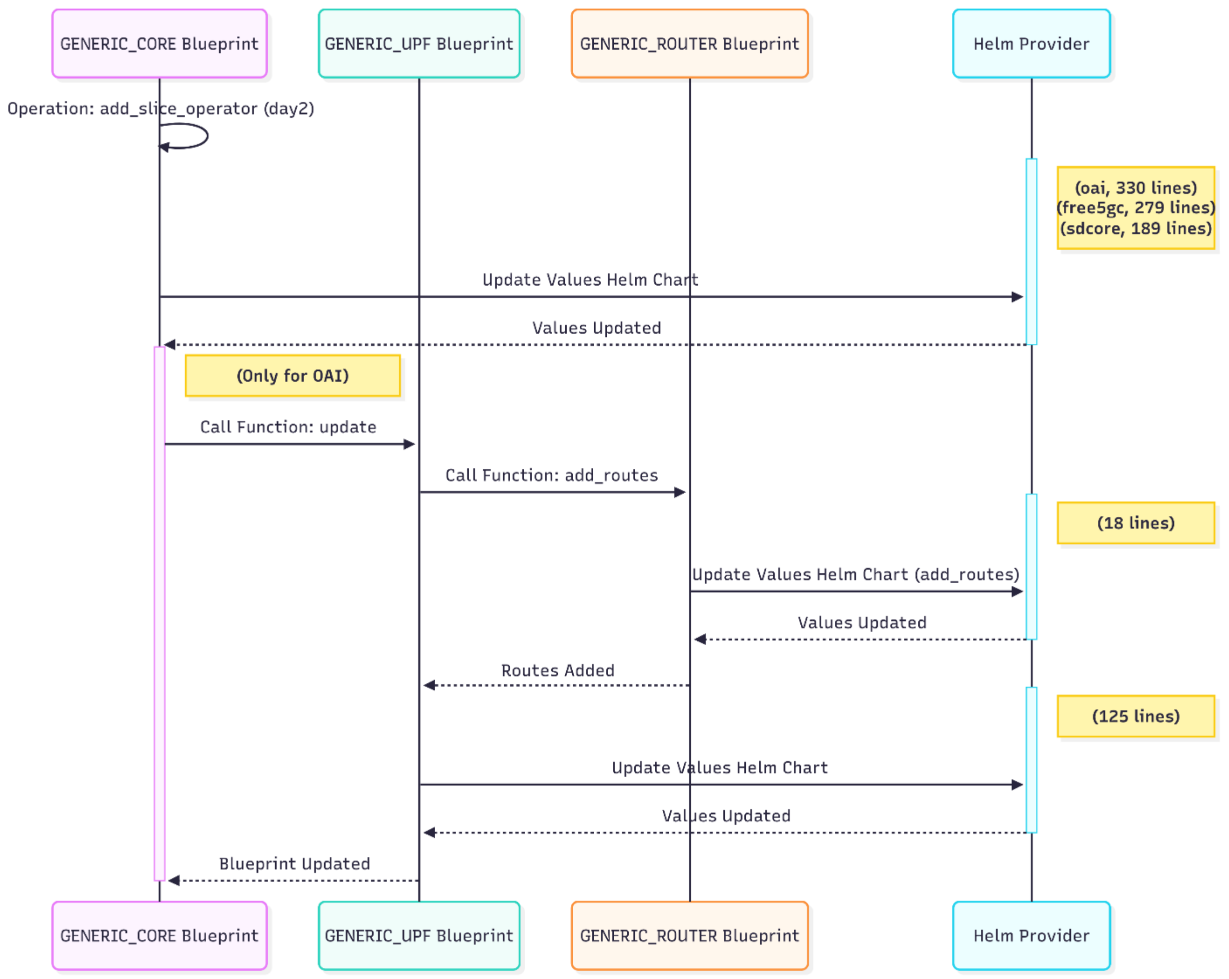


Figure A-5: Operations required to add a slice with the UPF provisioned in a pod.

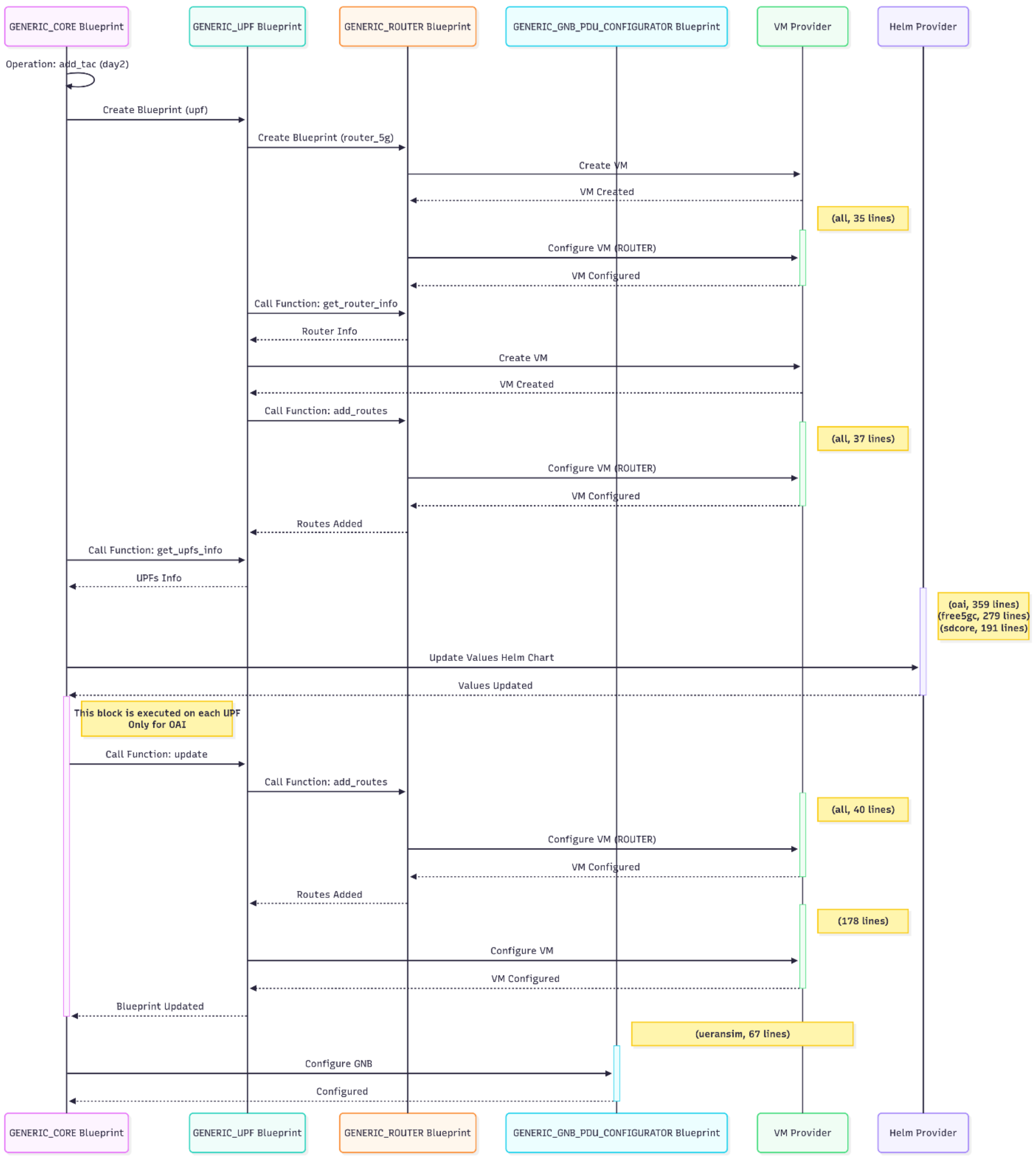


Figure A-6: Operations required to add a TAC with the UPF provisioned in a VM.

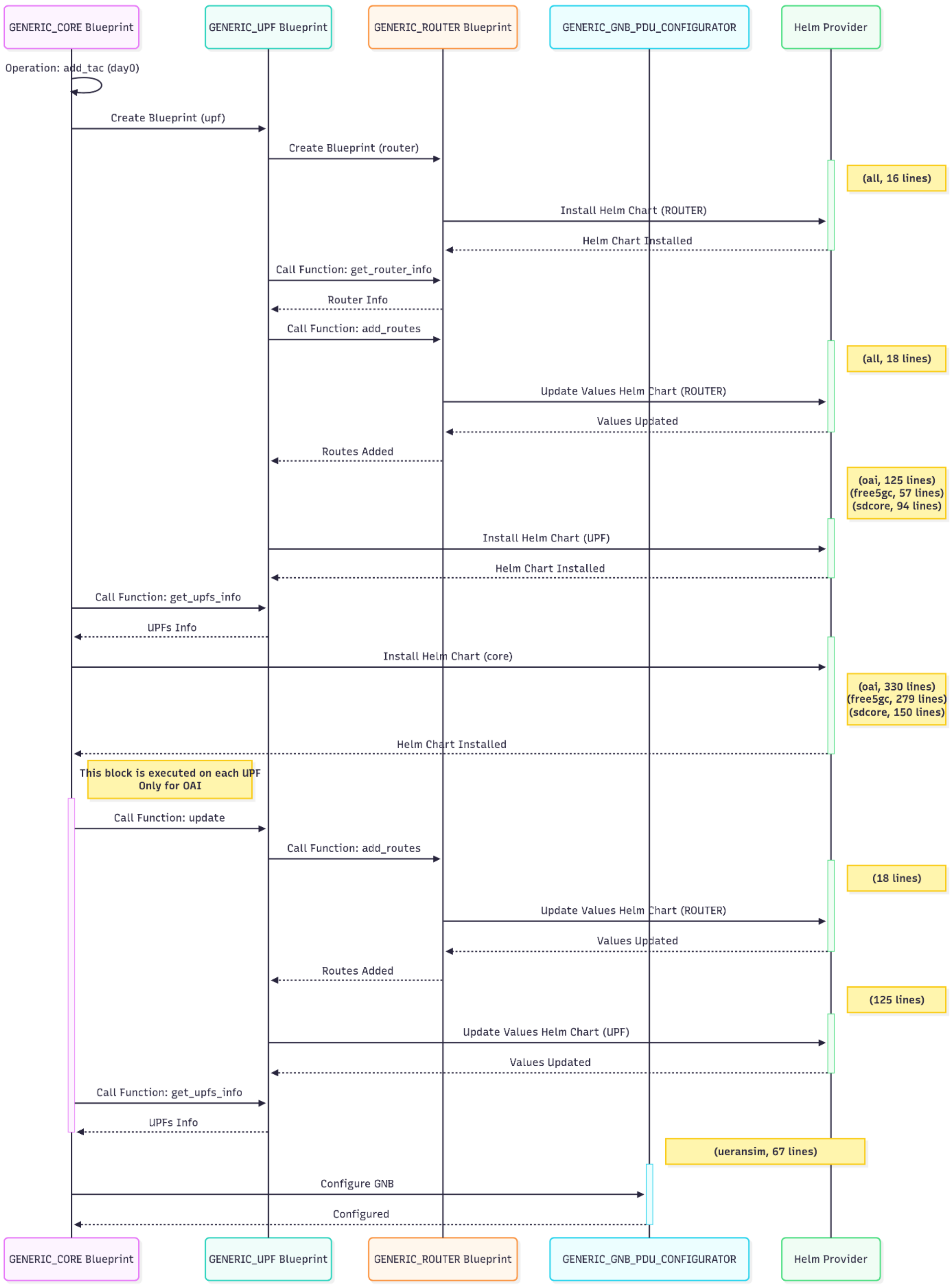


Figure A-7: Operations required to add a TAC with the UPF provisioned in a pod.

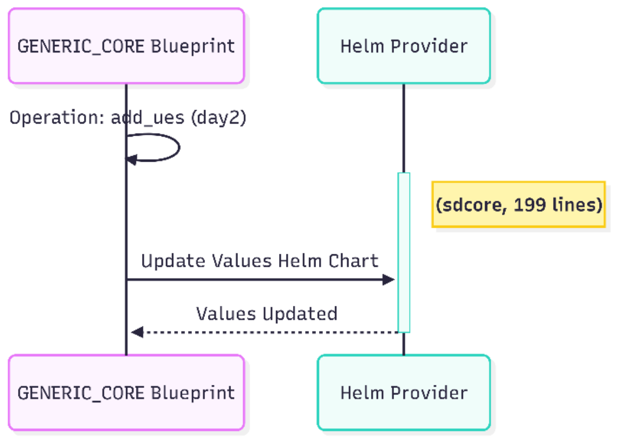


Figure A-8: Operations required to add a UE.

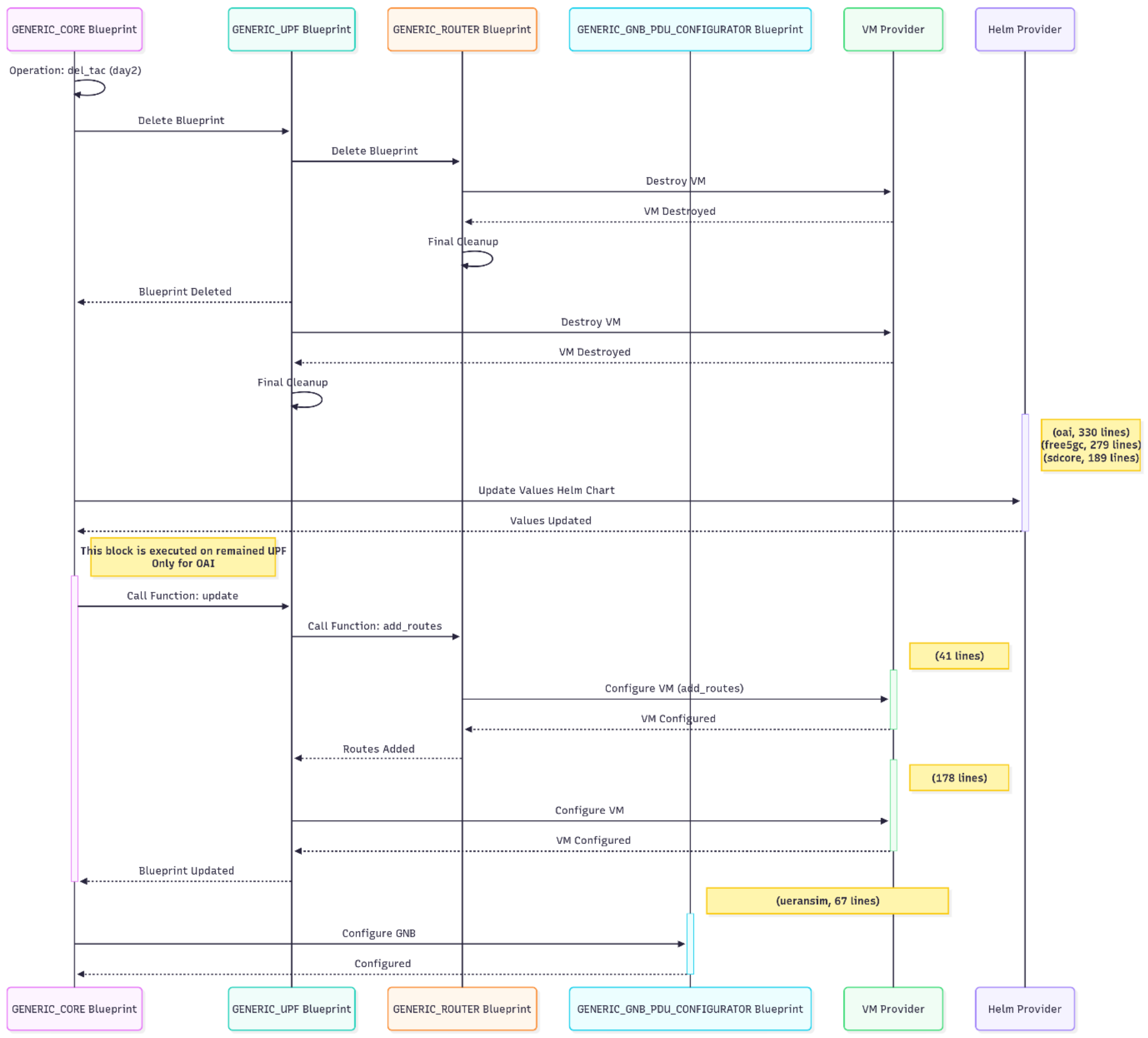


Figure A-9: Operations required to delete a TAC with the UPF provisioned in a VM.

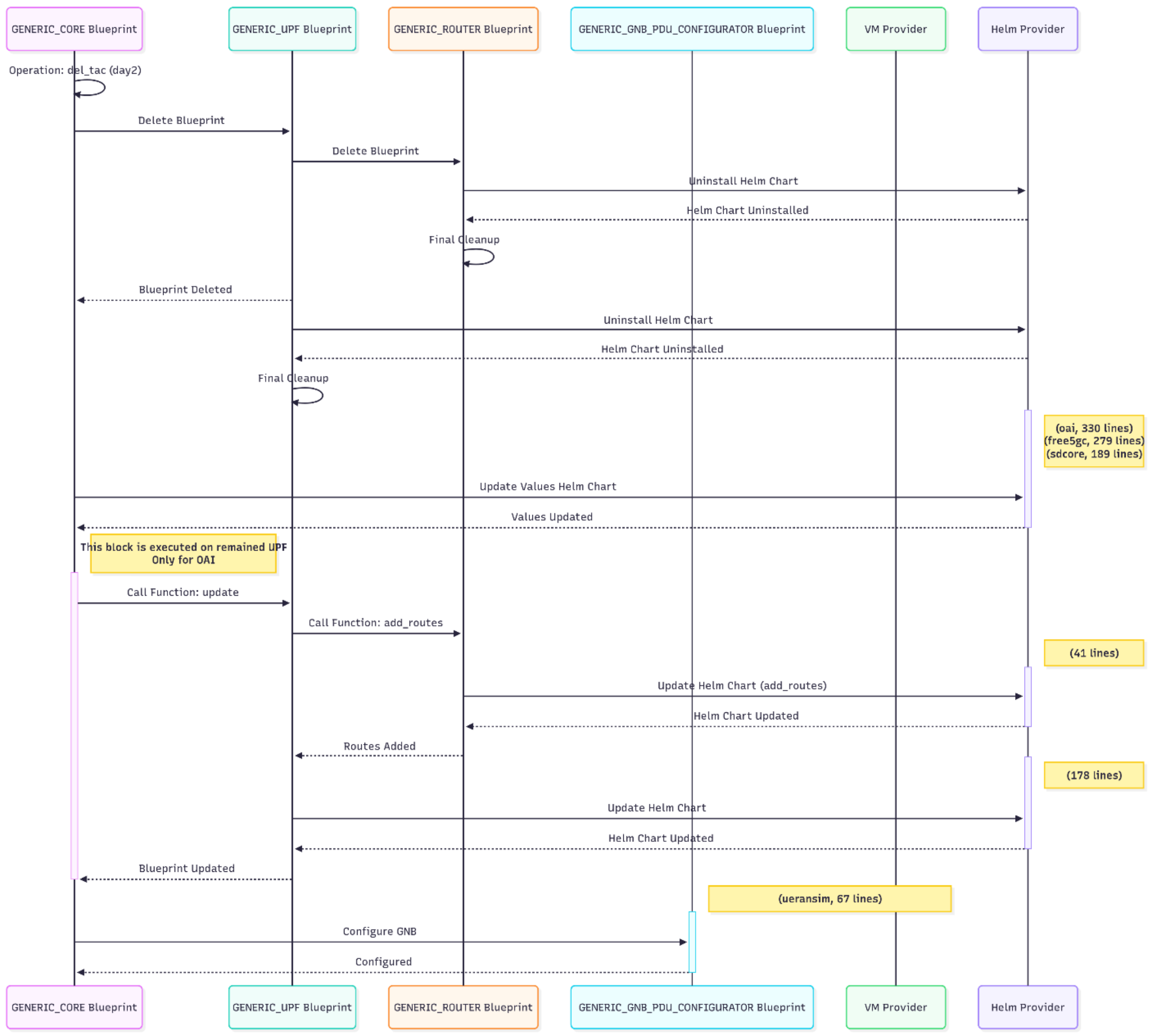


Figure A-10: Operations required to delete a TAC with the UPF provisioned in a pod.

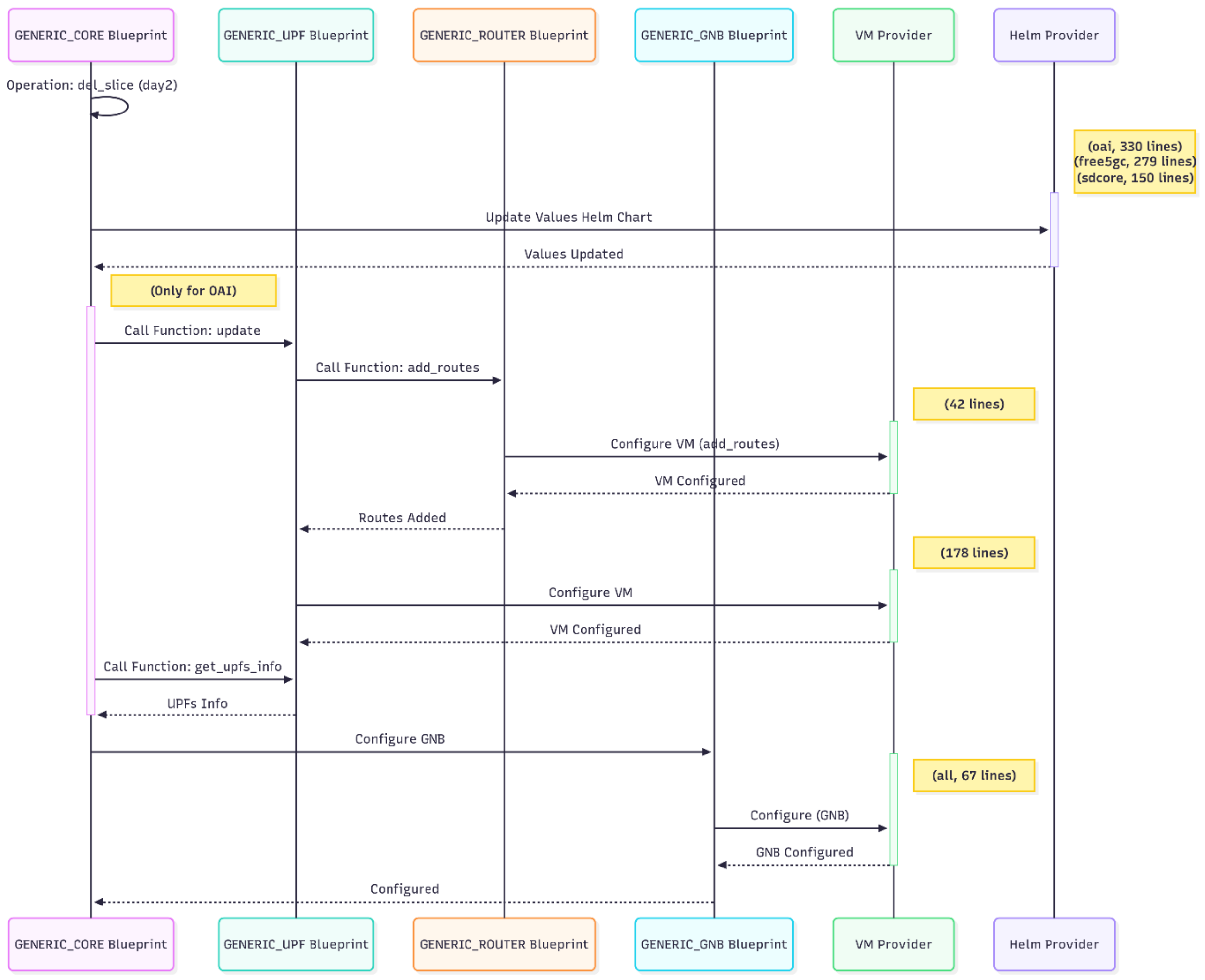


Figure A-11: Operations required to delete a slice with the UPF provisioned in a VM.

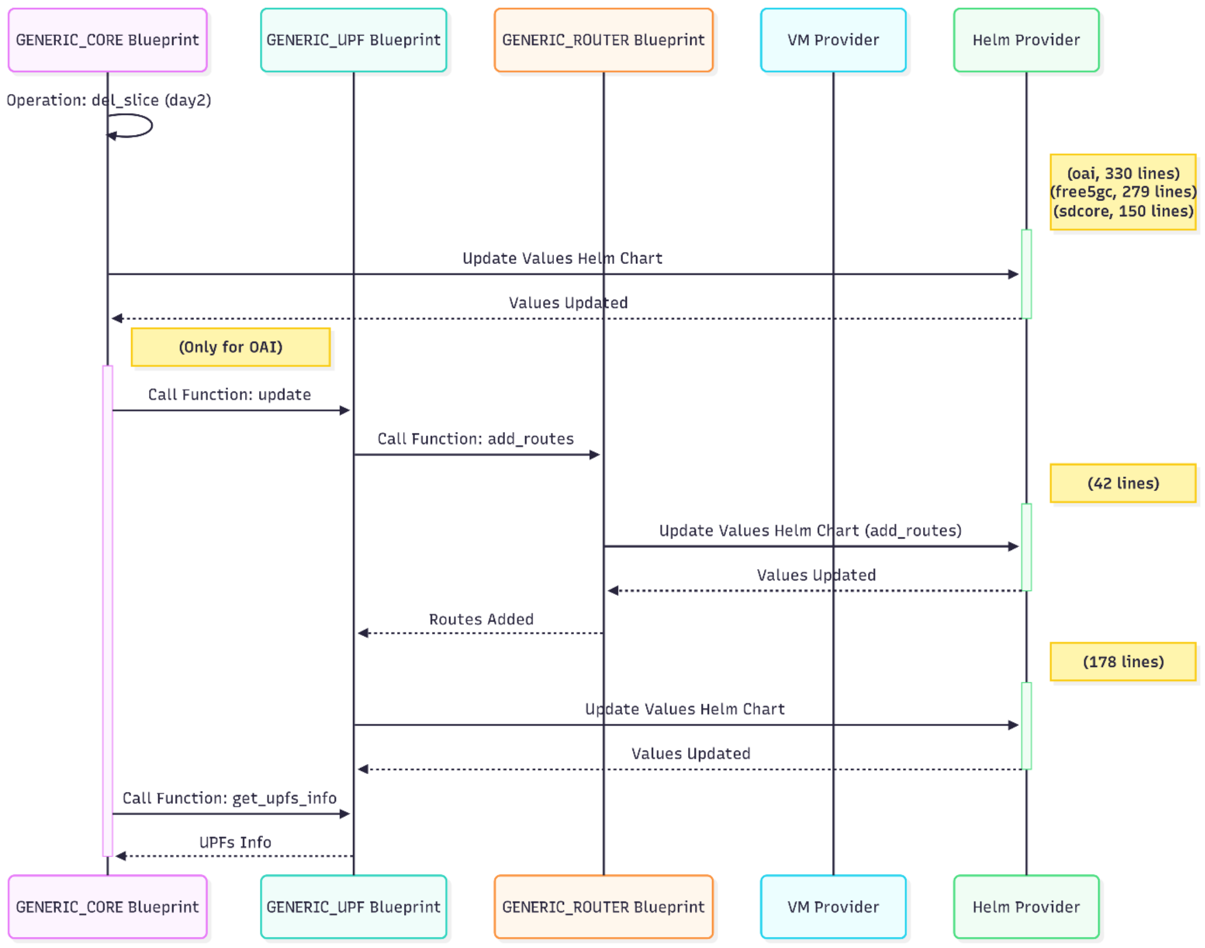


Figure A-12: Operations required to delete a slice with the UPF provisioned in a pod.

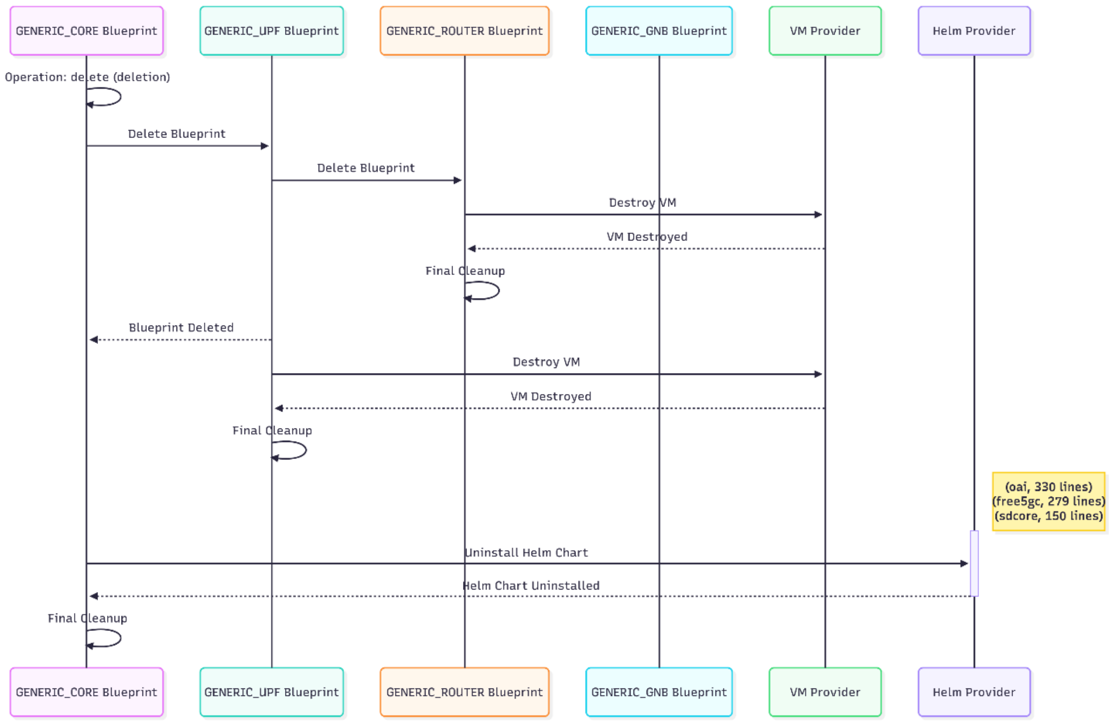


Figure A-13: Operations required to delete a core with the UPF provisioned in a VM.

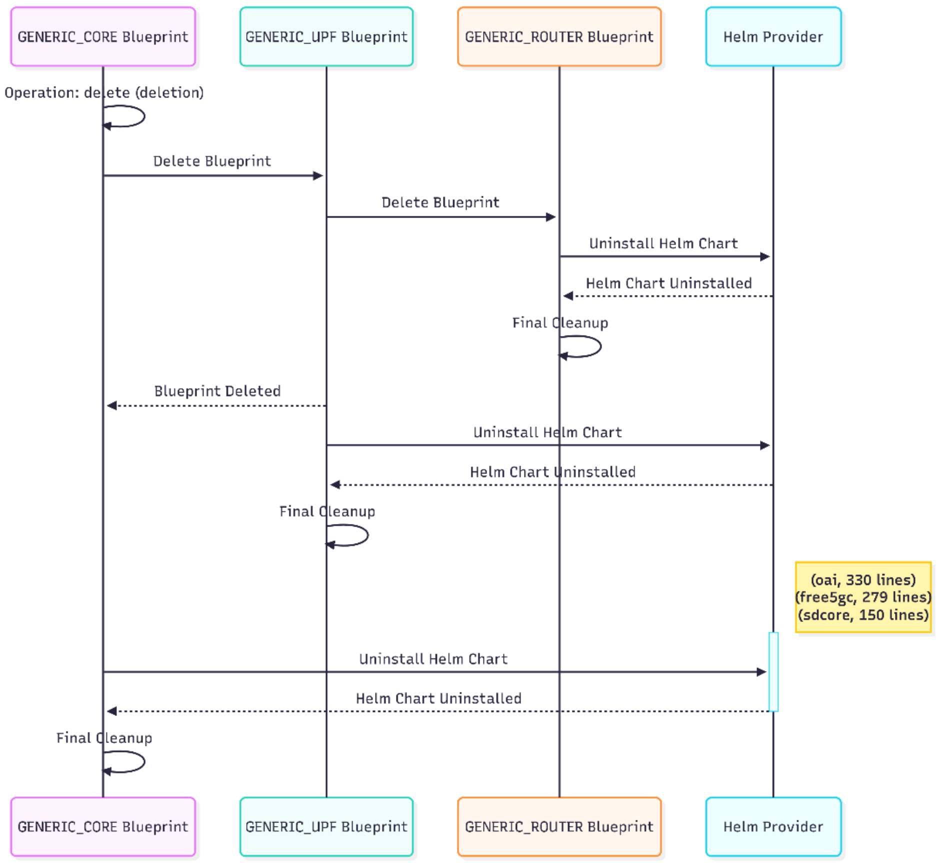


Figure A-14: Operations required to delete a core with the UPF provisioned in a pod.