



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A platform for efficient and observable management of multi-tenant Neutral Host deployments

TESI DI LAUREA MAGISTRALE IN
INGEGNERIA INFORMATICA
COMPUTER SCIENCE AND ENGINEERING

Author: **Andrea Caravano**

Student ID: 10708260

Advisor: Prof. Marco Mezzavilla

Co-advisors: Prof. Antonio Capone, Prof. Eugenio Moro

Academic Year: 2025-26

Abstract

As the geopolitical and regulatory landscape evolves, the requirement from Mobile Network Operators of transparent, efficient and auditable systems become more and more relevant. In a world where operational and computational efficiency of resources are a crucial milestone, Neutral Hosts play a fundamental role in the development and deployment of mobile radio networks.

This work proposes an open-source implementation of an efficient and observable platform for distributed multi-tenant Neutral Host deployments.

To do so, we leveraged and partially customized some of the most industry-leading solutions in the realm of efficient and distributed 5G deployments.

Thanks to ANSIBLE, we proposed a paradigmatic shift from common telecommunications-oriented deployments by building a declarative and customizable KUBERNETES cluster which minimizes computational and energetic footprints exploiting both boot-time and run-time optimizations.

Then, we built on the exceptional capabilities enabled by the open-source 5G ecosystem, focusing, in particular, on OPEN5GS and OPENAIRINTERFACE. Thanks to the privileged access to the newly-developed infrastructure as a whole, we were able to close the visibility gap in existing Neutral Host deployments, by enabling extensive inspection and control capabilities for hardware, network and radio metrics and parameters.

The experimental evaluation phase allowed us to conclude this path by integrating a power governance and traffic control framework for multi-tenant setups which further validate the chosen implementation strategies.

The resulting solution is readily applicable to a wide-variety of use-cases, such as real-time performance and SLA monitoring, anomaly and abuse detection, users and tenants billing cycles.

Keywords: Open-Source 5G, Computational Efficiency, Neutral Host, Observability Platform, Multi-Tenant Architecture

Abstract in lingua italiana

Con l'evolvere del contesto geopolitico e regolatorio, la richiesta da parte degli operatori radio mobili per sistemi trasparenti, efficienti ed ispezionabili è diventata sempre più rilevante. In un mondo in cui l'efficienza operativa e computazionale delle risorse sono un obiettivo cruciale, gli host neutrali giocano un ruolo fondamentale nello sviluppo e implementazione delle reti radio mobili.

Questo lavoro di tesi propone un'implementazione open-source di una piattaforma efficiente ed ispezionabile per i setup distribuiti e multi-tenant nel contesto degli host neutrali.

A tal fine, abbiamo utilizzato e parzialmente personalizzato alcune delle soluzioni più riconosciute nell'industria nel contesto delle implementazioni efficienti e distribuite in ambito 5G.

Grazie ad ANSIBLE, abbiamo proposto un cambiamento paradigmatico dalle soluzioni più comuni orientate alle telecomunicazioni, costruendo un cluster KUBERNETES dichiarativo e personalizzabile, che minimizzi l'impatto computazionale ed energetico, sfruttando ottimizzazioni all'avvio e nell'esecuzione del sistema.

Successivamente, abbiamo sfruttato le possibilità offerte dalle eccezionali soluzioni disponibili nell'ecosistema open-source 5G, concentrandoci, in particolare, su OPEN5GS e OPENAIRINTERFACE. Grazie all'accesso privilegiato alla infrastruttura appena sviluppata, nel suo complesso, siamo stati in grado di colmare il vuoto creatosi nell'ambito della osservabilità nelle implementazioni di host neutrali esistenti, abilitando estensive possibilità di ispezione e controllo per le risorse e metriche hardware, di rete e radio.

La fase di valutazione sperimentale ci ha consentito di concludere questo percorso integrando uno schema di riferimento per l'amministrazione del consumo energetico e del traffico per i setup multi-tenant, che ci ha permesso di validare ulteriormente le strategie implementative scelte.

La soluzione così risultante è immediatamente applicabile a una varietà di contesti e casi d'uso differenti, tra cui: valutazione prestazioni e monitoraggio dei livelli di servizio offerti in tempo reale, rilevazione di anomalie e abusi, fatturazione di utenti e tenant.

Parole chiave: Open-Source 5G, Efficienza Computazionale, Host Neutrale, Piattaforma Di Ispezionabilità, Architettura Multi-Tenant

Contents

Abstract	i
Abstract in lingua italiana	iii
Contents	v
Introduction	1
0.1 An architectural revolution: Open RAN	1
0.2 A new player: the Neutral Host	2
0.3 Thesis Contribution: closing the visibility gap	3
0.4 Thesis Outline	4
1 Background and state of the art	7
1.1 The O-RAN ecosystem	7
1.1.1 Motivations for an architectural shift	8
1.1.2 RAN architecture evolution	8
1.2 Virtualization and orchestration in Mobile Networks	11
1.2.1 Virtual Machines: a brief overview	11
1.2.2 Containerization: a brief overview	13
1.3 Containerization technologies and use-cases	14
1.3.1 Linux Containers	14
1.3.2 Docker	14
1.3.3 Kubernetes	15
1.4 A primer on Kubernetes	15
1.4.1 Control Plane	15
1.4.2 Worker Nodes	16
1.4.3 Abstractions	16
1.4.4 The Consensus Problem	17
1.4.5 Networking in Kubernetes	17

1.5	Reproducibility in system images	18
1.5.1	File-system Hierarchy Standard	19
1.5.2	A concrete example: OSTree	19
1.5.3	The Infrastructure as Code paradigm	20
2	A computationally efficient 5G deployment	23
2.1	An enterprise-ready solution: Red Hat OpenShift and OKD	23
2.1.1	Declarative machine configuration	23
2.1.2	CoreOS	24
2.1.3	Setting up an OKD cluster	25
2.2	A custom solution	30
2.2.1	Requirements	30
2.2.2	Kubernetes-oriented distributions	31
2.2.3	Linux distributions	32
2.2.4	Declarative idempotency: the Ansible paradigm	33
2.3	Recreating a middle ground: the Ansible Playbook	35
2.3.1	Requirements	35
2.3.2	Choosing a kernel flavour	36
2.3.3	Choosing a Kubernetes flavour	38
2.3.4	Container Network Interfaces	39
2.3.5	Multus CNI for 5G fronthaul	40
2.3.6	A brief overview: SR-IOV and DPDK	45
2.3.7	High Availability and Virtual IPs	45
2.4	System tuning for telecommunications workloads	48
2.4.1	Boot-time Kernel optimizations	48
2.4.2	Memory management	50
2.4.3	Run-time operating system tuning	50
2.4.4	Network stack tuning	51
2.4.5	Kubernetes resource management	53
3	A Neutral Host management platform	57
3.1	A primer on Kubernetes orchestration	57
3.1.1	Static declarative manifests	57
3.1.2	The Helm package manager	58
3.2	The Open-Source 5G Ecosystem	60
3.2.1	A quick overview at srsRAN	60
3.2.2	Open5GS	61
3.2.3	OpenAirInterface	62

3.2.4	FlexRIC	62
3.3	Deploying the Core Network	63
3.3.1	Structural overview	63
3.3.2	Authentication and key management	65
3.3.3	Interacting with Sierra Wireless UEs	68
3.3.4	Network slicing	70
3.4	Traffic policy enforcement	71
3.4.1	Linux Traffic Control	71
3.5	Deploying OpenAirInterface	76
3.5.1	Hardware paradigms: Ettus Research USRPs	77
3.5.2	Observability through Vector	79
3.5.3	The sink: HTTP Webhook	83
3.5.4	Deploying FlexRIC	87
3.6	The identity resolution problem	91
3.6.1	Advanced connection management and infrastructure monitoring . .	93
3.7	A live dashboard for end-users	96
3.7.1	PostgreSQL database	97
3.7.2	Redis key-value store	99
3.7.3	End-user interface	100
4	Experimental evaluation and use-cases	105
4.1	System profiling and energy tuning on AMD platforms	105
4.1.1	Model-Specific Registers for Power Control	105
4.1.2	AMD Host System Management Port	106
4.1.3	Telemetry and profiling via AMD μ Prof	107
4.2	Practical effects of CPU governors in a 5G deployment	109
4.2.1	A primer on CPU governors in Linux	109
4.2.2	The two extremes: performance vs powersave	110
4.3	A power governance framework for the Neutral Host management platform	115
4.3.1	Escaping the Pod	115
4.3.2	The staircase state machine	118
4.3.3	Multi-tenant energy billing	119
4.3.4	Deployment and security	120
4.3.5	A complete experimental execution	121
4.3.6	End-user interface	124
4.4	Use-cases	125
4.4.1	User and tenant billing	125

4.4.2	Real-time performance and SLA monitoring	126
4.4.3	Performance correlation	127
4.4.4	Anomaly and abuse detection	127
4.4.5	Spatial user mapping and positioning	128
5	Conclusion	131
5.1	Future Works	132
5.1.1	Positioning Reference Signals and Digital Twin paradigm	132
5.1.2	Official integration with open-source solutions	134
5.1.3	Machine Learning for proactive control	135
5.1.4	Adapting to novel hardware platforms: ARM/RISC-V	135
	Bibliography	137
A	Appendix A	141
A.1	Code publishing notes	141
A.2	Live simulative demo	141
	List of Figures	143
	Acknowledgements	145

Introduction

The widespread adoption of portable all-in-one telecommunication devices, the commercial deployment of Fifth-Generation (5G) and the research focus towards Sixth-Generation (6G) networks are driving the telecommunications field.

Mobile Network Operators (MNOs) play a fundamental role in such a growth and, in return, an increasingly-high amount of needs have emerged: high throughput for multimedia streaming and real-time experiences, low-latency communications, industrial and automotive control systems, low-cost connectivity for IoT sensors and microcontrollers.

External pressures are also driving newly-introduced paradigmatical shifts towards open and more easily controlled software and hardware ecosystems.

In fact, as users require more and more from their mobile networks, the complexity of supporting them both from an architectural and an economical standpoint grows accordingly.

On the other hand, as the geopolitical and regulatory landscape evolves, an increasing variety of concerns become more and more important: **supply-chain resilience, vendor lock-in, data management and control** and, first and foremost, the need for **transparent and auditable systems**.

As a result, MNOs and regulators are now more and more incline to favour interoperable and more inspectable systems.

0.1. An architectural revolution: Open RAN

Open RAN (O-RAN) is the industry response to these needs: O-RAN splits the Radio Access Network (RAN) into distinct **functional blocks that can be run as standard virtualized software** on Commercial Off-The-Shelf (COTS) hardware.

Communication among components happens using pre-defined and standardized interfaces and protocols, ensuring that different suppliers can interoperate.

The industrial principles proposed by O-RAN:

- **Open disaggregation:** the Radio Unit (RU), the Centralized Unit (CU) and the Distributed Unit (DU) are split in separate functional blocks.

Such paradigm allows for optimizations in function placement: time-sensitive processing can remain close to the Radio Unit, while non real-time tasks can be centralized in edge or cloud facilities.

- **Standards-based compliance:** allowing vendors to test their solutions against the standard in an open and more competitive environment.
- **Demonstrated interoperability:** ensuring disaggregated elements work together as a fully functional system, including all needed performance and security requirements.
- **Implementation neutrality:** standardized fronthaul and management APIs drastically reduce vendor lock-in and provides the basis for an environment where vendors can compete on components such as radio hardware, RAN software implementation and orchestration tools.

A crucial element of O-RAN success is **RAN programmability**: the RAN control plane exposes APIs for completely custom control logic and automation.

A RAN Intelligent Controller (RIC) offers two main operating modes:

- **Near-real-time control loops:** can tune radio parameters on sub-second timescales. They are generally referred to as **xApps**.
- **Non-real-time logic:** can be used to perform policy enforcement, analytics and even Machine Learning tasks. They are generally referred to as **rApps**.

0.2. A new player: the Neutral Host

The technical potential of a powerful and disaggregated RAN architecture lets new players enter the market with their offer.

Among those, the one that gained more traction is the **Neutral Host**: a specialized third party that owns, deploys and manages a shared physical RAN and attached computational resources that is then leased to multiple MNOs (also referred to as *tenants* in this space) [2].

The first consideration is an economical one: this model lowers both entrance and maintenance costs for new MNOs, by exploiting shared resources.

In many applications, in fact, duplicated or redundant infrastructure is impractical or inefficient:

- **Large and dense areas:** deploying hardware and software for each interested MNOs in largely dense environments (like sports palace or exhibitions) is often an inefficient choice.
- **Low-investment areas:** areas where investments for mobile networking coverage are stagnant gain significant advantage from the deployment of a shared infrastructure among MNOs, that would instead find difficult to expand their coverage to such zones.
- **IoT and private applications:** enterprise campuses can both provide private and public network access to their guests and their IoT infrastructure, sharing the same physical resources.

The picture is now clearer: when open interfaces, virtualization and RAN intelligence are combined, the neutral host can mix components from multiple vendors, instantiate network slices and enforce traffic control policies that isolate tenant traffic and guarantee specific per-tenant properties.

Virtualization allows the neutral host to map each tenant workload to a pool of compute resources that can be dynamically adapted to varying network conditions and user requirements, while enforcing control policies for capacity, priority and Quality of Service (QoS) assurance.

0.3. Thesis Contribution: closing the visibility gap

From a tenant perspective, sharing physical and virtualized infrastructure introduces specific and non-negotiable requirements:

MNOs and, more in general, shared network deployments must preserve:

- **Per-user accountability and metrics analysis:** this allows for standard billing, customer support, management and network analytics.
- **Service level enforcement:** control mechanisms that allow enforcement of service levels (throughput, availability) and ensure fairness among users scheduled traffic.
- **Hardware transparency:** providing resource utilization and hardware transparency guarantees Neutral Host users visibility on the complete chain that delivers the service to their customers.

- **Cost control:** by allowing Neutral Host users to fine-tune their resource usage, they also gain the ability of planning their usage proactively, based on their business and economic analysis.
- **Security and compliance features:** including data audits and dispute resolution processes [41].

Alongside the development of a solution to such requirements, this thesis builds on the resource sharing paradigm, by designing a specifically-tailored **Neutral Host deployment** that aims at exploiting the widest range of software and hardware optimizations made available on most architectures, therefore not relying on technological solutions made available by a specific hardware vendor.

This deployment is built on a customized **Debian GNU/Linux**-based environment that also applies the **Infrastructure as Code** paradigm, by being completely replicable and parameterizable, thanks to the usage of **ANSIBLE**, an automation tool-set that ensures, among others, the property of **idempotent** execution on multi-node setups.

0.4. Thesis Outline

In the rest of this thesis, we will study, structure and practically implement an efficient open-source observability platform for Neutral Host deployments:

- **Chapter 1, Background and state of the art:** provides the underlying theoretical foundation to build our platform.
We will detail the O-RAN ecosystem, outlining the architectural shift and its evolution.
We will then move to virtualization and containerization in the context of mobile networks, with a brief overview of the enabling technologies.
Finally, we will explore KUBERNETES, immutable distributions and the Infrastructure as Code paradigm.
- **Chapter 2, A computationally efficient 5G deployment:** describes the path towards the use of Infrastructure as Code tools in the context of mobile networks deployments, tuning both Kernel and run-time parameters.
We will first study OPENSIFT and OKD, observing their core features and architectural advantage to prepare our customized solution on a general purpose GNU/Linux distribution.
Finally, we will implement our **ANSIBLE PLAYBOOK**, outlining its high-level requirements and providing a declarative path to reach them, focusing, in particular, on

resource management and networking.

- **Chapter 3, A Neutral Host management platform:** presents the core contribution of this work.

We will initially prepare the ground for the deployment of the most used open-source solutions in the 5G space: OPEN5GS, OPENAIRINTERFACE and FlexRIC.

We will then implement OPEN5GS and manage the interaction with commercial UEs.

Finally, we will get into the core of the platform, allowing for traffic policy enforcement, radio and performance metrics analysis and identity resolution.

- **Chapter 4, Experimental evaluation and use-cases:** takes into analysis the practical effects of the chosen implementation strategy.

We will first detail energy tuning and profiling on AMD platforms, focusing on the use of official solutions for power control and energy-wise inspection.

Then, we will cover the use of CPU governors in the realm of a 5G deployment, outlining its relevance for the reduction of concrete power usage in absence of advanced control mechanisms.

Finally, we will explore our custom power governance framework, built around the knowledge gained from the Neutral Host management platform, and we will immediately put it to use to conduct an experimental campaign on our solution.

1 | Background and state of the art

This chapter provides the fundamental elements underlying an O-RAN deployment on COTS hardware, outlining, at first, the core blocks of most distributed systems and containerized workloads.

We will first cover a typical research-oriented O-RAN test bed on COTS hardware, focusing our attention on architectural aspects and underlining the adhesion to its aforementioned core principles.

Then, we will explore a typical **containerized environment** in this space, focusing our attention on its main players: DOCKER, KUBERNETES and multi-node clusters.

We will then be ready to build on such foundation by carefully dissecting some of its core fundamentals and integrate our own customized ones, thanks to our **Ansible Playbook**.

Finally, we will cover the underlying technologies to the **Neutral Host Management Platform** and its inner mechanisms.

1.1. The O-RAN ecosystem

As anticipated, O-RAN reshaped the RAN industry and, in general, the deployment of new-generation mobile radio networks, by providing a more intelligent, open, virtualized and fully interoperable Radio Access Network.

Since 2018, the O-RAN ALLIANCE is the responsible body for its standardization, driven by growing consortium of MNOs, technology vendors and academic institutions operating in the RAN industry, that has since gained a more competitive and vibrant ecosystem [37].

1.1.1. Motivations for an architectural shift

- **SDN-inspired wireless networks:** since 2008, Software Defined Networking (SDN) is a key paradigm of switched networks.
O-RAN builds on this inspiration, transferring its concepts to mobile radio networks.
- **Management and control complexity on different timescales:** complexity of mobile radio networks has grown due to technological advancements and network architectures. Management and control can be better performed with a centralized view and disaggregated elements to be controlled.
This also allows for better programmability, at different time scales ranging from seconds down to milliseconds, as anticipated.
- **Virtualization and COTS hardware:** the industry is pushing towards the direction of implementing network functionalities as software modules over general purpose hardware (COTS).
The use of COTS hardware allows economical scale and reusability, while also enabling ease of maintenance.
This novel approach naturally extends to cloud-first scalable environments.
- **Interoperability:** the architectural disaggregation and centralized control allows for easier combination of components.
Multi-vendor setups become possible and foster competitiveness, while discouraging single ones to lock their customer in their ecosystem (vendor lock-in).

1.1.2. RAN architecture evolution

Fourth-Generation RAN

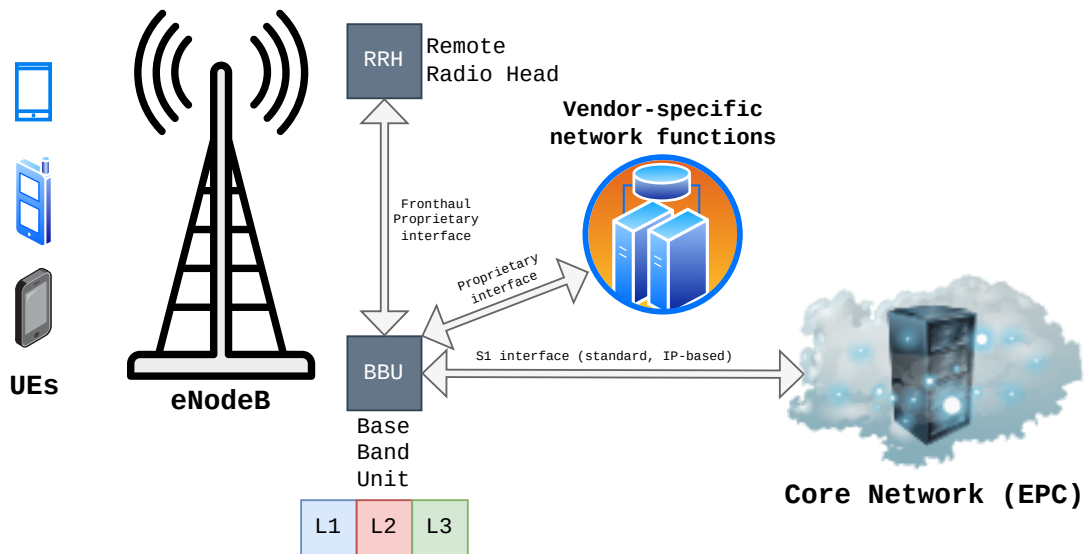


Figure 1.1: A schematic view of the old 4G RAN architecture, which shows the broad usage of proprietary interfaces

In the Fourth-Generation (4G, LTE) architecture, the main RAN element is the eNodeB. Functionally, it implements the first three layers of the protocol stack and interfaces to a centralized core (EPC) for mobility and session management.

The eNodeB is therefore the result of a **commercial offer**: it combines the radio unit (RRH) with baseband processing in a single vendor-supplied solution, deployed directly at the chosen site.

The vendor becomes, therefore, the reference point for maintenance and upgrades, since it also provides proprietary network functions.

The O-RAN paradigm

The 5G specifications formalize logical disaggregation of the RAN entity (the gNB) into separated functional blocks, defining:

- **Centralized Unit (CU)**: it ultimately splits into User Plane (IP packets coming from and to the user) and Control Plane (signalling).
- **Distributed Unit (DU)**: typically virtualized on COTS hardware employing dedicated hardware accelerators.

- **Radio Unit (RU)**: it hosts low-level (physical), radio transmission and processing.

The standardization body (3GPP) also introduces a family of functional split options, enumerated from 1 to 8, with further refinements like 7.1 or 7.2, that centralizes or further split some inner elements of the architecture.

O-RAN builds on such foundation by:

- Standardizing open and interoperable interfaces between the O-RU, O-DU and O-CU
- Introducing the **RAN Intelligent Controller (RIC)** that supports custom control logic.

As anticipated, it is typically split into two operating modes: the Non-Real-Time RIC (operating on timescale greater than one second) and the Near-Real-Time RIC (operating instead on smaller scales, around tens of milliseconds).

- Defining a Service Management and Orchestration Platform framework (SMO): a centralized structure providing overall management and orchestration of the O-RAN configuration, policy enforcement, performance and lifecycle.
- Introducing an O-Cloud reference for hosting virtualized RAN elements.
It ultimately resolves into a collection of hardware and software stacks.

We now need to standardize new open interface definitions, that will allow O-RAN elements to communicate:

- **Open Fronthaul**: supports communication between the O-RU and the O-DU, including management and interoperability tests.
- **E2** interface: connects the **near-RT RIC** and RAN nodes (CU/DU), provides telemetry and parametric resource management.
- **O1** interface: connects the **SMO** and the **non-RT RIC** for network operations, configuration and maintenance, adopting a combination of REST and HTTP-based APIs.
- **A1** interface: connects the **non-RT RIC** and the **near-RT RIC** (or **SMO**), conveying policies, optimization model updates and parameter negotiation.

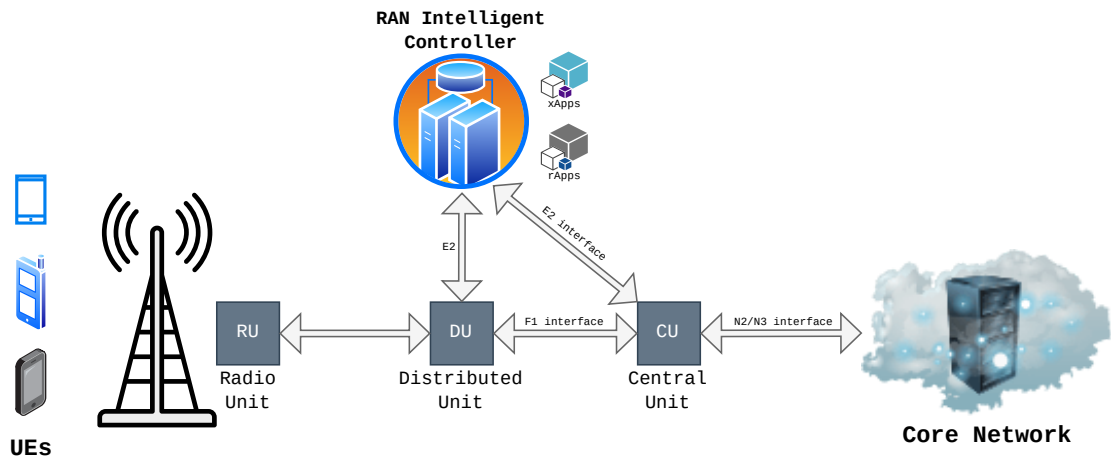


Figure 1.2: A brief overview at RAN disaggregation in O-RAN, outlining standardized interfaces

The Sixth Generation (6G) specifications will also converge development efforts by consolidating the most used technological combinations and build on them.

1.2. Virtualization and orchestration in Mobile Networks

As anticipated, a pivotal element in the O-RAN paradigm is virtualization of O-CU, O-DU and RIC components, alongside supporting network functions on COTS hardware, following the O-Cloud reference.

1.2.1. Virtual Machines: a brief overview

Popek and Goldberg defined a virtual machine as:

- An efficient, isolated duplicate of the real system that can run a commodity OS (system-level virtualization).
- Based on a virtual machine monitor that relies solely on direct execution

Which ultimately leads to requiring:

- **Fidelity:** equivalence of behaviour with the real machine.
- **Safety:** the virtual machine cannot override the VMM's control of virtualized resources.

- **Efficiency:** programs should only show minor decreases in speed.

Virtual machines are used to consolidate and partition hardware usage, reacting to varying workloads and reduce redundant and administrative hardware costs, while also providing fault tolerance.

Instead of employing two physical machines at 40% capacity, a single physical machine can be used at 80% capacity to run multiple workloads simultaneously, resulting in fewer, more powerful machines.

From a deployment perspective, they provide a **standardized infrastructure that simplifies software distribution** for structured environments, while also providing tools for versioning or rollback strategies (e.g. Snapshots).

Terminology

A host system is the operating system on which virtual machines run, while the guest system is the operating system running on top of the virtual machine.

The Virtual Machine Monitor (VMM) or Hypervisor mediates access to physical resources and ensures isolation.

The Virtual Machine Monitor can either run on bare metal without any OS abstraction (type 1 hypervisor) or in the context of another operating system (e.g. KVM).

Virtualization can be largely categorized into:

- Software-based virtualization: relying on de-privileging, meaning the guest OS operates at a lower privilege level than the host one.
The hypervisor intercepts and emulates privileged instructions from the guest OS via a process generally referred to as **trap-and-emulate**.
- Hardware-based virtualization: introduced as a solution to bypass the problems related to de-privileging (e.g., excessive faulting and higher complexity of operating modes). A notable example is represented by Intel VT-x, which introduced state management and efficient virtual memory management.

A tangible example

Kernel-based Virtual Machine (KVM) is an open source virtualization technology built into the Linux Kernel, allowing for Linux to work as a hypervisor, via hardware-based virtualization.

Quick EMUlator (QEMU) is commonly used as a KVM-based VMM, which, therefore, overlooks emulation of devices (e.g., virtual disk, virtual network card) and direct hardware passthrough (e.g. physical GPU or network card) thanks to the VirtIO interface.

Proxmox Virtual Environment (PVE) is a commonly used GNU/Linux-based hypervisor using QEMU-KVM as its enabling technologies.

1.2.2. Containerization: a brief overview

Containers are a way to isolate a set of processes and make them think they are the only ones running on the machine.

Similarly to virtual machines, the part of the machine they are able to see may be a subset of overall total available resources.

However, **containers are not virtual machines**:

- Processes running inside a container are normal processes running on the host kernel.
- There is no guest kernel or operating system running inside the container.
- The kernel is shared with the host, so you cannot run an arbitrary operating system inside a container (in most cases, Linux).

Namespaces

The fundamental mechanism driving container isolation on Linux are namespaces.

A namespace changes and reduces the set of resources that a process belonging to such namespace can access or manipulate.

- **PID namespaces**: allow for independent process trees.
Processes in a child namespace cannot interact with processes in the parent tree.
- **Unix Timeshared System namespaces**: give each process tree its host and domain name.
- **Mount namespaces**: provide different views of the file-system, similarly to the mechanism employed by **chroot**.
- **Network namespaces**: isolate network stacks, including routes and networking interfaces.
- **User namespaces**: map processes inside a container to different user and group identifiers than the ones on the host.

The advantage of sharing a common kernel while isolating processes and integrating development dependencies creates the perfect scenario for deploying RAN functions as microservices running on containers.

1.3. Containerization technologies and use-cases

1.3.1. Linux Containers

Linux Containers (LXC), in 2008, built on such foundation by allowing multiple isolated Linux-based environments on a host, sharing all the same kernel.

The LXC framework provides a comprehensive user-space tool-set (libraries, templates and bindings) to manage existing Linux Kernel capabilities.

Pivotal elements to LXC are, of course, namespaces and control groups (**cgroups**), that determine what a process can use in terms of physical resource management, allowing the host to limit and prioritize usage of the underlying hardware.

The result is a container that mimics a traditional virtual machine as closely as possible, without the hypervisor overhead.

LXCs also carry over a complete init system (e.g. **systemd**) and adopts standard system daemons, while also allowing for user management.

1.3.2. Docker

In 2013, DOCKER revolutionized the way of using containerization by providing a **standardized, developer-friendly abstraction layer** over these primitives.

DOCKER introduced the concept of **Container image**: an immutable, layered file-system bundle containing the application code, runtime, system tools, libraries and configuration files all-together.

The DOCKER architecture is actually client-server based, with the DOCKER Client communicating with the DOCKER Daemon, the entity responsible for building, running and distributing container images, which in turn uses standard containerization runtimes.

Container images are built as layers: multiple containers running on the same base environment will share the same layers, optimizing storage and provisioning strategies.

1.3.3. Kubernetes

While DOCKER excels at running containers on a single host node, enterprise-grade solutions, such as a distributed 5G Core Network or RAN requires spanning many containers across clusters of physical machines.

Managing deployment, scaling, load-balancing, networking and fault tolerance manually across such a complex distributed system is prone to inevitable failures.

KUBERNETES (sometimes referred to as K8s) emerged as a popular solution for **container orchestration** [3].

It operates on a declarative configuration model: administrators **declare the desired state of the infrastructure** (e.g. number of replicas and hardware passthrough).

A specific control loop constantly observes the actual state of the system and takes automated decisions to reconcile it with the desired declared state, effectively automating operations.

As we will observe, ANSIBLE will follow a similar path, which will, however, omit automation by default.

1.4. A primer on Kubernetes

A KUBERNETES cluster is fundamentally divided in two co-operating planes: the Control Plane (the master of the system) and the Data Plane (often referred to as Worker Nodes, where effective workloads will run).

1.4.1. Control Plane

- **API Server:** the front-end of the Control Plane, exposing the KUBERNETES API. Internal and external communication all pass through this component.
- **etcd:** a highly-available, consistent key-value store used as KUBERNETES storage for all cluster data and state.
Consistency and availability are designed following the typical distributed systems RAFT consensus algorithm.
- **Scheduler:** watches for newly created PODS (atomic units of scheduling) having no assigned node and selects one of them to run on based on resource requirements and constraints, also considering (anti-)affinity specifications: an advanced scheduling features dictating which nodes a POD can be assigned to and how PODS should be

placed relative to each other.

- **Controller manager:** runs controller processes that perform the aforementioned reconciliation loops.

1.4.2. Worker Nodes

- **Kubelet:** an agent running on each node of the cluster.
It ensures that containers are running in a POD according to the specifications provided declaratively.
- **Kube-Proxy:** a network proxy running on each node, maintaining network rules that allow network communication to PODS from inside or outside the cluster.
- **Container Runtime Interface (CRI):** the plugin interface that allows the KUBELET to use a variety of container runtimes (e.g. containerd or CRI-O) without needing to recompile KUBERNETES.

1.4.3. Abstractions

KUBERNETES models workloads through a set of objects:

- **Pod:** roughly anticipated earlier, a POD is the atomic unit of scheduling.
A POD encapsulates one or more containers sharing storage, a network namespace and lifecycle.
In our use case, a POD may contain Core Network functions or RU-interacting workloads.
- **ReplicaSet:** an abstract structure ensuring the specified number of POD replicas are running as declared.
- **Deployment:** allows for declarative updates to PODS and ReplicaSets, providing update and rollback strategies.
- **DaemonSet:** ensures that a set of nodes runs a copy of a specific POD, ensuring node-level operations execution like logging, monitoring and network plugins management.
- **StatefulSet:** used for stateful applications, providing ordering and guarantees for a set of PODS.

1.4.4. The Consensus Problem

In the realm of Distributed Systems, the Distributed Consensus Problem has always been a relevant challenge: getting multiple independent nodes to agree on a single, unified state of the system, even in presence of **non-byzantine** failures (e.g., network partitions, packet loss, node crashes).

As per the CAP theorem, when a network partition occurs, a system must choose between Consistency and Availability.

KUBERNETES chooses Consistency, managed by its foundational key-value store, **etcd**.

etcd achieves this consistency using the RAFT Consensus Algorithm: RAFT is designed around the concept of a replicated state machine and relies on a strong leader approach.

The cluster elects a single Leader. To commit a configuration or state update, the leader must successfully replicate the log entry to a quorum (majority) of nodes.

1.4.5. Networking in Kubernetes

KUBERNETES imposes a strict networking model: all PODS must be able to communicate with all other PODS without using Network Address Translation (NAT) and the IP that a POD sees must be the same that all other PODS see.

To achieve this, KUBERNETES uses the **Container Network Interface** (CNI).

The CNI specifications define a standard API for network plugins that configure networking interfaces: when the KUBELET is tasked with starting a POD, it delegates the network setup to the active CNI plugin (which can also be a combination of more than one).

The plugin is responsible for creating and assigning a virtual network interface into the POD's networking namespace, assigning then an IP address from a pre-defined Classless Inter-Domain Routing (CIDR) block and updating routing tables on the host.

Common CNI plugins include **Calico**, **Flannel**, **Cilium** and **Multus**.

Traditional CNI plugins, however, introduce software abstractions and virtual bridges, which add to latency and computing overhead constraints.

Multus and SR-IOV

A 5G RAN (and, in general, specialized Telecommunications applications) require strict latency requirements, which ultimately leads to dedicated hardware interfaces able to

handle high throughput and very small timescales.

This requires specialized networking primitives within KUBERNETES:

- **Multus:** KUBERNETES natively assigns only one network interface (often `eth0`) to a POD.
Multus CNI acts as a multiplexer, enabling the attachment of multiple networking interfaces to a single POD.
Secondary interfaces provided by Multus can be mapped to real hardware interfaces, tailored for low latency applications.
- **SR-IOV:** alongside Multus, the SR-IOV CNI plugin allows exploiting the SR-IOV hardware feature of advanced NIC, which can therefore appear as single physical NICs (more specifically, it applies to PCIe devices).

1.5. Reproducibility in system images

The concept of state entropy in operating systems refers to the gradual divergence of a system's configuration from its initial, known baseline state.

In a traditional GNU/Linux distribution (e.g., Debian, Ubuntu, Fedora), the root file-system (`/`) is mounted in read-write mode.

System administrators, package managers (e.g., APTITUDE, `dnf`) and applications commonly mutate the state by, for example, writing configuration files to `/etc`, placing binaries in `/usr/bin` and updating shared libraries in `/lib`.

If hardware fails, restoring the exact software environment from a fresh setup requires a complete historical log: dissimilarities will result in what is known as a configuration drift.

The most common solutions to achieve reproducibility of system images are:

- **Monolithic immutability:** Locking down the file-system to prevent changes.
- **Functional immutability:** Treating the OS as a mathematical function of its inputs, achieving pure reproducibility without locking standard file system paths.
- **Declarative idempotency:** Leaving the file-system mutable but utilizing an external engine to continuously pull the current system configuration and push changes.

1.5.1. File-system Hierarchy Standard

As per the requirements and guidelines for file and directory placement under UNIX-like operating systems, the following paths are defined, originating from the root file-system (/) [24]:

- `/bin`: Essential user command binaries, for use by all users.
- `/dev`: Device files.
We will get in touch with device files more in detail when deploying our gNodeB to a USB-based USRP (Software Defined Radio).
- `/etc`: Host-specific system configuration.
- `/home`: User home directories.
- `/lib`: Essential shared libraries and kernel modules.
- `/opt`: Add-on application software packages.
- `/run`: Run-time variable data.
- `/tmp`: Temporary files.
- `/usr`: Shareable, read-only data.
- `/var`: Variable data files: administrative and logging data, transient and temporary files.

1.5.2. A concrete example: OSTree

In an immutable distribution, the vast majority of the operating system, including binaries, shared libraries and core components, are mounted as read-only.

Some GNU/Linux distributions, including FEDORA's atomic spins, the FLATPAK engine and some RED HAT's CENTOS derivatives use OSTREE, which operates following the same ideas of GIT, the ubiquitous **version control system**.

When an immutable OS boots, the bootloader (most commonly, GRUB), points to a specific deployment within the `ostree` repository [6].

A deployment is a **complete, bootable file-system tree checkout**, in which:

- `/usr`: is mounted as read-only. Any attempt to write here will fail at the Virtual File System (VFS) level.

- `/etc`: is mounted as read-write. During OS upgrades or image modification, a GIT-styled merge will happen.
- `/var`: is mounted as read-write. This is where containers, logs and databases will live.

Because the whole OS image is defined as a specific commit hash in the `ostree`, updating or modifying the system is an atomic operation. The live, currently-running system is **never modified**: the update daemon will fetch relevant objects and write them to the repository, creating a new (default) bootloader entry.

The system is then reloaded to complete checkout on the new tree head. Rollback is therefore as easy as pointing the tree head to the previous `ostree` commit.

DOCKER images work similarly, even though resulting images are the result of a linear execution and does not stem from versioning trees and instead relies on layers.

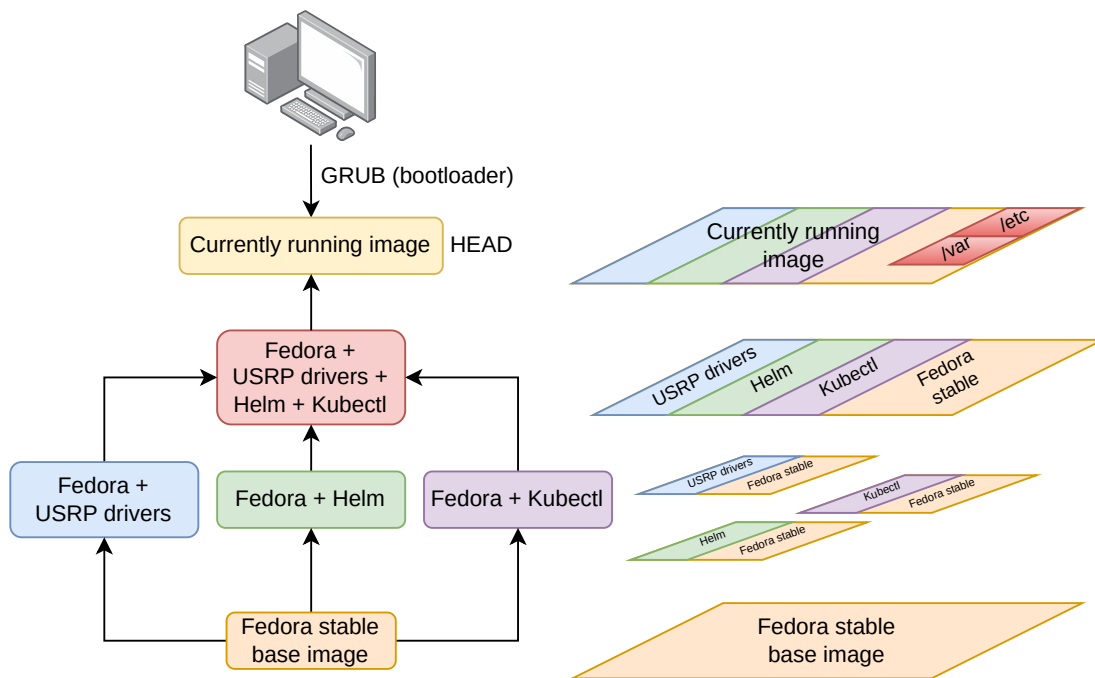


Figure 1.3: The construction of a system image

1.5.3. The Infrastructure as Code paradigm

Infrastructure as Code (IaC) is the engineering practice of managing and provisioning computing infrastructure through human-readable definition files, rather than physical hardware configuration or interactive configuration tools.

IaC tools generally revolve around the following paradigms:

- **Imperative (Procedural):** the code explicitly defines a list of steps required to achieve a resulting state.
- **Declarative:** the desired final state is defined, the IaC tool is then responsible for delineating the set of steps required to transition the system from the current state to the desired state.

In the following chapters, we will analyze in more detail `ANSIBLE`, a declarative IaC tool.

2 | A computationally efficient 5G deployment

Being a research-oriented project, we will always choose open-source software solutions whenever available, which will allow us to analyze their working mechanisms in detail and build around them.

The entire architecture will rely on a **single-node, bare-metal KUBERNETES cluster**.

While the typical use case for KUBERNETES is a three (or more) node cluster, as anticipated when discussing the distributed consensus problem, a single node deployment will still allow us to appreciate the capabilities of the resulting solution and will be immediately ready for higher scale setups.

2.1. An enterprise-ready solution: Red Hat OpenShift and OKD

Red Hat OPENSIFT Container Platform (OCP) and its community open-source project, OKD, represent an integrated ecosystem designed to abstract and automate management of a single or multi-node KUBERNETES cluster.

In the telecommunications space, OPENSIFT and OKD are increasingly popular solutions because of their overall complete compatibility with most common requirements for real-time network processing.

2.1.1. Declarative machine configuration

Unlike standard KUBERNETES distributions, where the underlying operating system (typically, a GNU/Linux distribution) and the container orchestrator are decoupled, OPENSIFT (and OKD) merges these two layers via specifically-designed abstractions, the Cluster Version Operator (CVO) and the Machine Config Operator (MCO) [1].

The CVO constantly monitors the cluster's current state against a declarative release image containing all core cluster elements.

OPENSHIFT heavily relies on the **Operator pattern**: an operator is an application-specific controller that extends the KUBERNETES API by using Custom Resource Definitions (CRDs).

In OPENSHIFT, every system component (container registries, networking plugins, ingress controllers, logging and monitoring routines) are governed by an internal operator.

For telecommunications workloads, OPENSHIFT introduces **specialized operators**, such as the Node Tuning Operator (NTO), which uses the **tuned** daemon to manage network card parameters, kernel and memory management properties.

2.1.2. CoreOS

FEDORA COREOS (FCOS) is the underlying host operating system to OKD (while OPENSHIFT Enterprise uses RED HAT ENTERPRISE LINUX COREOS, RHCOS) [13].

FCOS is a minimalist, custom-built operating system optimized exclusively for running containerized workloads.

Escaping immutability: rpm-ostree

As anticipated, traditional Linux distributions treat the file-system as a mutable environment, where administrators install, modify and delete packages arbitrarily via package managers like APTITUDE (**apt**) or **dnf**.

FCOS relies, instead, on the aforementioned **ostree**.

The core file system resides in a read-only directory (**/usr**), while mutable configurations are confined to **/etc** and **/var**, as per the FHS standard. The operating system is therefore deployed as a single file-system image.

When a modification or a package layering is requested via **rpm-ostree**, the system does not modify the live runtime environment. It instead creates a **new bootable file-system tree**, combining the base image with the requested changes.

We will need to extend FEDORA COREOS when measuring the Core Processing Unit (CPU) power consumption and metrics.

2.1.3. Setting up an OKD cluster

While a traditional installation procedure is generally the preferred route for a typical desktop setup, setting up a multi-node cluster may involve configuring and managing many machines.

The typical choice for such a scenario is automated or semi-automated setups, where most of the low-level configuration of the machine, including, for example, disk partitioning, networking, DNS configuration, hostnames and users is done through **automation**.

The OKD installation program offers the following methods to deploy a cluster [14]:

- **Interactive:** The web-based Assisted Installer is used. This is the suggested approach for deploying clusters in networks directly connected to the internet. It provides a set of pre-defined defaults and performs validation checks throughout the process.
- **Local Agent-based:** It provides many of the benefits from the Assisted Installer, configuration is performed via the command-line interface. This is the preferred choice for disconnected environments, allowing also for use of self-provided registries.
- **Automated:** The installation program uses a controller for provisioning, allowing both connected and disconnected environments.
- **Full control:** A more traditional installation, which allows for the maximum flexibility and customization.

In the following, we will analyze the transition from a traditional installation to an assisted one, highlighting our first obstacles.

Traditional installation

A traditional OKD installation requires, as of June 2026, a minimum of 4 CPU cores, 16 GB of RAM and 120 GB of storage [10].

When deploying our ANSIBLE PLAYBOOK, we will lower down such requirements. However, while lower requirements are enough for many use-cases, in a general setting, these are also the **most common baseline requirements for a complete 5G-ready stack**.

The server must also have access to the internet or a local registry. The server must have a DHCP reservation or a static IP address for the KUBERNETES API, ingress route and cluster node domain names.

A DNS server must be configured to resolve the IP address for the following Fully-Qualified Domain Names (FQDNs).

Table 2.1: Required DNS records for OKD

Usage	FQDN	Description
Kubernetes API	api.<cluster_name>.<base_domain>	Standard A/AAAA/CNAME record. Must be resolvable by both external and internal clients to the cluster.
Internal API	api-int.<cluster_name>.<base_domain>	Standard A/AAAA/CNAME record. Must be resolvable by in- ternal clients to the cluster.
Ingress route	*.apps.<cluster_name>.<base_domain>	Wildcard A/AAAA/CNAME record. Must be resolvable by both external and internal clients to the cluster.

A **local DNS server** allowing for custom overrides is suggested (e.g., `dnsmasq`).

To proceed, another host running a standard desktop operating system is suggested. The `podman` project should be installed.

Then, the OKD client (`oc`) must be downloaded from the OKD project reference repositories, alongside the FEDORA COREOS (FCOS) ISO image.

We finally come in contact with the most distinctive implementation detail of OPENSIFT and OKD: **declarative configuration and personalization**.

```

1  apiVersion: v1
2  baseDomain: '<domain>'
3  compute:
4    - name: worker
5      replicas: 0
6  controlPlane:
7    name: master

```

```

8   replicas: 1
9   metadata:
10    name: '<name>'
11   networking:
12    clusterNetwork:
13     - cidr: 10.128.0.0/14
14      hostPrefix: 23
15    machineNetwork:
16     - cidr: 10.0.0.0/16
17    networkType: OVNKubernetes
18    serviceNetwork:
19     - 172.30.0.0/16
20   platform:
21    none: {}
22   bootstrapInPlace:
23    installationDisk: '/dev/disk/by-id/<disk_id>'
24   pullSecret: '<pull_secret>'
25   sshKey: |
26    <ssh_key>

```

In which,

- **The base domain** is the cluster domain name (e.g. 'polimi.lab').
- **Compute** replicas are set to 0, while **Control Plane** replicas are set to 1.
By default, KUBERNETES strictly separates Control Plane nodes and Runner/Compute nodes. The combination of these two settings will make our single node (which will, of course, also run as a control plane) schedulable (i.e., accept to run PODS).
- **The machineNetwork** is the physical network for nodes. This is the common network from which nodes have access to local resources (e.g. storage) or the internet.
- **The clusterNetwork** is the internal network to running PODS.
Each POD will receive an IP address from this pool. Each node of the cluster will receive a /23 (512 addresses) network for their containers.
- **The serviceNetwork** is the network used by KUBERNETES services.
It allows PODS to communicate with each other using a unique, stable address. We will familiarize with KUBERNETES services when setting up the Neutral Host Management Platform.
- **The remaining network and disk configuration** details are coherent with tra-

ditional GNU/Linux distributions install procedures.

After completion, ignition details are embedded into the FCOS ISO and finally, burn to a standard CD/DVD drive or written to a USB stick.

The resulting image will be the DISCOVERY ISO which will automatically trigger ignition, write the system configuration to the chosen disk and start installation.

On the administration node, the installation can be monitored via `openshift-install`.

After completion, the resulting `kubeconfig` YAML file will be written, allowing for management via the KUBERNETES CLI or the `oc` utility.

The `kubeconfig` is a small credential and routing wallet for the management of a KUBERNETES cluster: it contains API server URLs, Certificate Authorities, access token and private keys for access.

The standard placement for `kubeconfig` is in `~/.kube/config`, in a similar setting to SSH keys. Both the KUBERNETES CLI and `oc` utilities will default to such placement, which will allow us to configure a portable development environment, based on DOCKER.

Interactive installation via Assisted Installer

The OKD Assisted Install Service is the resulting composition of a set of pre-configured container images and variables, in a similar setting as KUBERNETES HELM packages, that we will encounter shortly when deploying our PLAYBOOK and platform.

The `sno.yaml` file is the main element here: it defines basic networking parameters, the COREOS image and OKD version to be used. It will be run using `podman` on the **administration node** (an external desktop PC, for example).

Hardware resources, DNS and network requirements are exactly like in the traditional installation route and a local DNS server is again suggested.

The administration node must be visible by all cluster nodes during installation, as the Assisted Installer web service will be used to complete configuration thanks to the DISCOVERY ISO.

Via the provided web page, we complete the same parameters we manually specified via the `install-config.yaml` file in the traditional method.

At the end of the installation procedure, if successful, the `kubeconfig` YAML file will be provided and kept stored through the Assisted Installer web service.

The `oc` utility and KUBERNETES CLI can be used exactly like in the traditional method.

A first obstacle

As of June 2026, the official guide on OKD documentation outlining the Interactive installation procedure through the Assisted Installer, which also provides a detailed sample of the procedure we roughly sketched above, states that **the resulting generated cluster will not conclude installation successfully or will encounter operational issues** [9].

The most relevant issue we encountered when testing both strategies is the impossibility of base distribution and OKD upgrade. An OKD upgrade can be forced by providing a fixed OKD image reference, but, even in case of completion, KUBERNETES abstractions and objects migration will fail.

Upgrading OKD and its base distribution (COREOS) is a **non-negotiable requirement for our use-case**: we will shortly introduce AMD μ PROF, a profiling suite from AMD which, in its recent versions, requires Linux Kernel version 6.0 or later, which, as of June 2026, a default OKD installation does not provide.

We also can't take the route of downgrading AMD μ PROF, since our workstation CPU, used throughout this work, **requires a recent version** of the profiling suite to produce a meaningful result.

A successful upgrade path

In the documentation sample, the administration host IP address is 192.168.0.141.

This is, of course, the first parameter to look for and adapt according to the setup.

Then, the default POSTGRESQL container image must be updated. The image referred to by the documentation is no more available and, even if retrieved from archival registries, should not be run on a production setting, since it has not been updated in several years.

The default image `quay.io/centos7/postgresql-12-centos7:latest` can therefore be replaced by `quay.io/sclorg/postgresql-12-c8s:latest`.

Then, COREOS base image, alongside OKD, needs to be updated. By default, OKD 4.12 and COREOS 37, from November 2022 is used.

A reasonable upgrade path is therefore the following:

```

1 OS_IMAGES: '[{"openshift_version":"4.20","cpu_architecture":"x86_64",
2   url":"https://rhcos.mirror.openshift.com/art/storage/prod/streams/c10s/builds/
3   10.0.20251023-0/x86_64/scos-10.0.20251023-0-live-iso.x86_64.iso",
```

```
4  "version":"10.0.20251023-0"}] '
5  RELEASE_IMAGES: ' [{"openshift_version":"4.20", "cpu_architecture":"x86_64",
6  "cpu_architectures":["x86_64"], "url":"quay.io/okd/scos-release@sha256:
7  c29eebdf042a002d0325bc94a8e93dab3872c6a247ac842907b735feaa076940",
8  "version":"4.20.0-okd-scos.13", "default":true}] '
```

This configuration will succeed in patching: the assisted installer web service will try to patch the FCOS ISO image to prepare it for the discovery procedure.

After installation, however, we observe that, even if the Linux Kernel version is now adequate for our use case, upgrades are still locked and, when forced, will result in a migration and reboot loop of the cluster.

Also, even if the laboratory workstation is more than capable of running heavy workloads, just running a baseline install of OKD, resulted in higher resource usage than expected and caused many of the initial tests, run on a lighter but still capable machine, to fail entirely.

We therefore **felt the necessity for a paradigmatic shift** and outlined our needs to prepare the ground for a new, more efficient solution.

2.2. A custom solution

2.2.1. Requirements

As we have learned, OKD and OPENSHIFT use standard GNU/Linux interfaces and applets, like **tuned** for system tuning.

The main value added by OKD is the **Infrastructure-as-Code system management**, which resembles the KUBERNETES static manifests and HELM packages.

We will inspect our possibilities keeping an eye on this matter, without compromising on performance, malleability, adaptability and generality of the result.

The resulting system should therefore be:

- **Configurable via code:** the result of the execution and the code should be inspectable and parameterizable, allowing for customization and adaptability to any cluster scale and underlying hardware configuration.
- **Compatible with custom utilities:** as anticipated when analyzing OKD, AMD μ PROF is a strict requirement to our solution: it will allow us to precisely measure

CPU power consumption and metrics.

- **Efficiently tuneable:** we would like the same degree of deep system tuning that OKD offers, exploiting real-time kernel support, core isolation and network parameter improvements.
- **Malleable:** we will need to build a Neutral Host Management Platform on our system, we would like it not to be an obstacle for our implementation.
- **Extensible:** many specialized network equipment (e.g., Software Defined Radios, Network Cards, Modems) will require to install drivers or Kernel modules to work. Our choice will also need to be capable to operate coherently.
- **General:** we would also like our system to employ generally-available solutions, that have minimal or no deviation from the standard and most commonly adopted strategies in the industry.

This goes directly in-line with the objective of a M.Sc. Thesis: using and building on generally available tools and knowledge, that can be then easily extended to any derived and proprietary one.

2.2.2. Kubernetes-oriented distributions

In recent years, many GNU/Linux distributions developed the objective of providing a specialized environment for running containerized workflows, such as DOCKER and KUBERNETES clusters.

Among those, the most relevant to our use-case are:

- **Talos Linux:** loses in generality and malleability, as most of the configuration and maintenance is done through a custom tool (`talosctl`).
To empathize on security concerns, TALOS also strips down many binaries typical of a GNU/Linux installation, locking down even SSH access [22].
- **Flatcar Container Linux:** uses the standard BUTANE IGNITION for first-boot setup, but does not allow for successive refinements using the same approach (if not using external tools).
Allows for many of our interest points, but loses on compatibility with custom utilities and practicality, since it does not have a built-in mechanism for boot on UEFI-based systems, since its main objective is a virtualized environment (e.g. PVE) [11].
- **Fedora CoreOS derivatives:** OPENSIFT and OKD are a part of this subset.

2.2.3. Linux distributions

The functional paradigm: NixOS

While image-based immutable distributions like COREOS solve the configuration drift problem by locking down the file system, they introduce operational friction (frequent reboots, heavy OS images and need for planned upgrade and maintenance schedule).

NIXOS approaches the problem from a radically different angle: **functional programming principles**.

In NIXOS, the operating system is not a composition of static images and layers. The OS is instead **the result of a mathematical function**.

Functional package management

In a traditional OS, installing a library will mainly involve writing to `/lib/`, for example. Invoking the library from an application will simply require using the FHS standard and retrieve it from `/lib/`.

The NIX package manager (and NIXOS in particular) does not rely on the FHS standard. Every package, configuration file and library will live in the NIX STORE, under `/nix/store`.

Derivations are the functional elements that populate the NIX STORE: they are a set of steps that the NIX package manager must adhere to whenever applying a configuration embedding such derivation. It specifies what components need to be built (compiled) or downloaded and determines their combined usage.

Paths in the NIX STORE will be the result of a hashing function, applied to a varying number of elements, depending on the specific derivation, alongside its name and version.

Declarative system state definition

In NIXOS, **the entire state of the machine** (including which packages are installed, which users exist, which disks are configured, SYSTEMD services) **is defined in a set of configuration files** (which often reside in `/etc/nixos/`) written in a functional programming language: the NIX language.

Applying the configuration produces an **evaluation of functional expressions and a consequent switch** of the system to the prescribed state.

The current system state is therefore itself a derivation, a specific entry in the NIX STORE.

Of course, **usability comes at the price** of heavy symbolic-linking and many executables need to be adapted to fit into this unique ecosystem.

However, many advantages are also made evident:

- **No immutability overhead:** Even if, of course, many differences apply, the NIXOS file-system is not immutable and the root file-system is technically standard.
- **Atomic rollbacks:** Every generation of the OS is just a different symbolic link pointing at a different derivation in the NIX STORE. Rolling back simply translates to pointing at a different derivation.
- **Complete reproducibility:** The set of NIX configuration files can be moved (or even synced to) a completely different physical server and still apply coherently. Of course, proper adjustments can be made to ensure hardware compatibility.

Sadly, NIXOS **loses in generality and compatibility:** while it is mostly based on standard GNU/Linux utilities, the NIX language is unique to the NIX package manager and NIXOS.

Also, even if AMD μ PROF and custom device drivers can be compiled and run on NIXOS through unofficial derivations [25], updates have broken compatibility in the past and adding a troubleshooting layer to our deployment does not feel adequate.

Lastly, many software artifacts need to be re-compiled from scratch or wrapped in an FHS layer to ensure proper compatibility with NIXOS.

2.2.4. Declarative idempotency: the Ansible paradigm

Both image-based immutable distributions and NIXOS require fundamentally altering the structure of the Linux file-system: they tend to break compatibility with many of the most common operational knowledge.

Since we need to build a solution which is as enterprise-ready as possible, we feel that adopting these would result in an unacceptable architectural risk from an enterprise standpoint.

We therefore would like to achieve **determinism and declarativity through Infrastructure-as-Code tools without altering the underlying mutable operating system** [32].

In recent years, the industry answered these needs with ANSIBLE: **a descriptive open-source automation language for any-scale IT environment**.

Embracing mutability

In an ANSIBLE-managed environment, the target system is a **completely standard, highly-mutable distribution** (e.g. Debian, Ubuntu, Fedora standard).

The file-system is the same one we are used to, with full read-write permissions based on users authorization as the FHS standard prescribes.

The system remains completely controllable, both remotely (via, for example, SSH) and physically through the terminal Command-Line Interface (CLI).

While the resulting environment is susceptible to configuration drifts, ANSIBLE counters this entropy by acting as an external enforcement engine.

Idempotency

The core engine of ANSIBLE relies on **idempotency**: an operation is idempotent if it produces the same result regardless of whether it is executed once or multiple times.

$$f(f(x)) = f(x)$$

The easiest comparison with shell scripts (`.sh`) is replication of actions:

Creating a directory through `mkdir /var/myscript` in an imperative shell script will fail when executed twice (as the directory already exists). While, of course, proper adjustments can be made to simple scripts (e.g. by using `mkdir -p /var/myscript` in this example), this approach is not general and cannot be applied indefinitely.

ANSIBLE takes this approach as its **mathematical foundation and builds a declarative automation language on it**.

Applying the same action in ANSIBLE just requires **declaring the desired final state**:

```
1 - name : Create the directory for my scripts
2   ansible.builtin.file:
3     path: /var/myscript
4     state: directory
5     mode: '0755'
```

Behind the scenes, the ANSIBLE engine will:

- Connect to the target node via SSH using provided keys/credentials.
- Dynamically generate a PYTHON script containing the specific module logic.

- Transfer the script to the target node.
- Execute the script on the target and inspect the current state of the file-system.
- Check whether the directory exists and permissions match. In such a case, the script will end immediately with no real change applied. If instead they are divergent, the script will apply required actions to converge to the desired final state.

ANSIBLE will not run continuously by default, but can be externally adapted to do so, which will make it work like a self-healing entity.

Administrators can therefore act on the state of the system just by altering the ANSIBLE PLAYBOOK and force the system into compliance with it.

2.3. Recreating a middle ground: the Ansible Playbook

We are now ready to exploit the knowledge we gained until now to build our own ANSIBLE PLAYBOOK: a set of actions that will be applied on the chosen set of machines to which our deployment will apply.

We will start **from a DEBIAN GNU/Linux based distribution**, like UBUNTU SERVER. Any other distribution based on DEBIAN will work without modifications to the PLAYBOOK.

Other distributions will instead require adapting workflows to match varying package managers (e.g. `dnf` instead of `apt`) and kernel tuning. Remaining actions can be kept unchanged.

2.3.1. Requirements

As we observed, ANSIBLE matches all the architectural requirements we expressed earlier. Now, we need to focus on the practicalities of building a 5G-ready setup starting from a standard GNU/Linux distribution, clearing out all relevant details during the process.

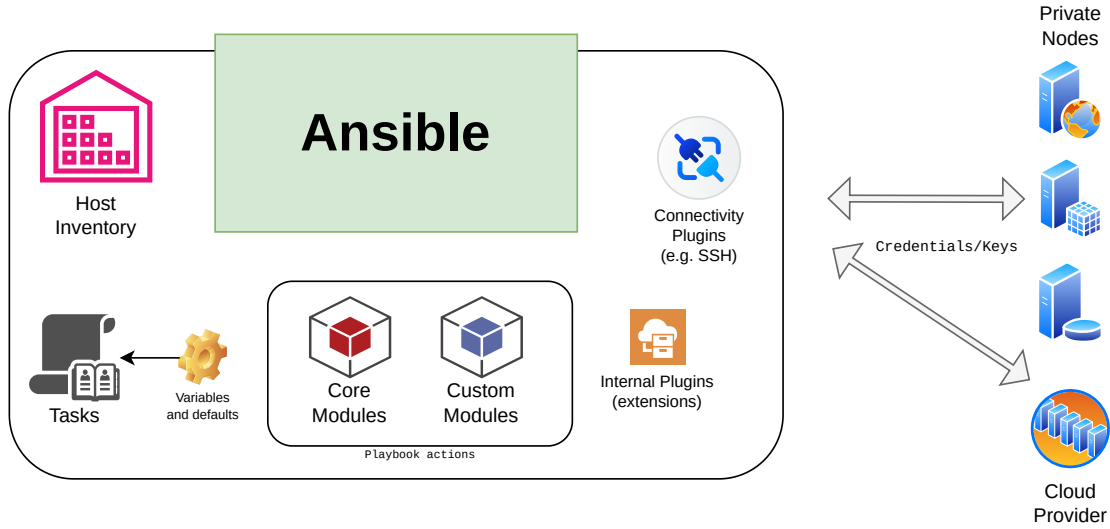


Figure 2.1: A summary of ANSIBLE's architectural model

We will therefore need to:

- Apply system tuning for RAN workloads
- Switch to the real-time kernel
- Bootstrap cluster nodes
- Configure network interfaces and modules
- Deploy a load balancer
- Bootstrap server and agent nodes
- Join additional server nodes to the control plane
- Apply manifests after cluster creation
- Finalize updates and dependencies for profiling

Let's tackle, one by one, these objectives. System tuning and energy management will have their own section, as they constitute one of the primary interest points in our solution.

2.3.2. Choosing a kernel flavour

In a standard 5G NR deployment, time is divided in slots. The typical duration of a slot is $0.5 \text{ ms} = 500 \mu\text{s}$

Within this 500 μs window, the OAI gNB (the DU in particular), must execute a significant pipeline of digital signal processing.

This generally includes:

- **Receiving samples** via the fronthaul interface (often at 10 or more Gb/s speeds).
- **Perform FFT** to move signals from the time domain to the frequency domain.
- **Channel estimation and equalization**
- **Parity check and decoding**
- **Passing the resulting packet (PDU) to the MAC layer for processing**

If the system scheduler pre-empts the DU process during this pipeline to run a background task, the process may miss the strict slot deadline.

Missing the deadline causes a dropped radio frame. The User Equipment (UE) perceives a failure, triggers retransmission which damages the effective throughput.

A primer on Linux Schedulers

The standard Linux Kernel and its Completely Fair Scheduler (CFS) are a masterpiece of general purpose computing.

The CFS is designed to maximize overall system throughput and fairness: it models an ideal multi-processor by maintaining a virtual runtime for every task.

The scheduler traverses an embedded tree data structure to find the task with the lower virtual runtime and grants it CPU access.

The result is excellent fairness for most workloads (like web servers, databases, etc.) but may not be ideal for 5G workloads. As anticipated, we would like the 5G DU to have strict priority over most tasks.

Linux provides adequate scheduling classes, like `SCHED_FIFO` or `SCHED_RR`. Tasks in these classes have higher priorities than any CFS task.

However, due to the inner workings of the Linux Kernel, some sections of kernel code are strictly non-preemptible. The total latency L_{TOT} experienced by task waking to up process an event is defined as:

$$L_{TOT} = T_{HW} + T_{IRQ} + T_{NON_PREEMPTIBLE} + T_{CONTEXT_SWITCH}$$

Where:

- T_{HW} is the hardware propagation delay.
- T_{IRQ} is the time taken to process the interrupt.
- $T_{NON_PREEMPTIBLE}$ is the time spent executing non pre-emptible code.
- $T_{CONTEXT_SWITCH}$ is the time taken to save and restore registers.

The acquisition of a SPINLOCK in the Linux Kernel disables pre-emption. Latency can therefore span over the slot budget and contribute to worsen throughput.

Linux as a RTOS

A Real-time Operating System (RTOS) is the perfect fit for this role: Linux achieves this by **applying the PREEMPT_RT patch** [31]. It guarantees that the $T_{NON_PREEMPTIBLE}$ is **bounded to a fixed value** (typically, around 20 μs).

SPINLOCKS are converted to MUTEXES, sleeping locks, allowing the CPU to run other tasks. Pre-emption is therefore no longer disabled: a high-priority task, like our 5G DU runs with deterministic and fixed time boundaries.

Behind the scenes, many other smaller optimizations are present, including, for example, threaded hardware interrupts, allowing for higher degrees of hardware control.

2.3.3. Choosing a Kubernetes flavour

We now need to address the orchestration layer. 5G network functions are, of course, highly-compatible with KUBERNETES-compliant platforms.

Since we now gained a higher degree of freedom, we can (and should) look more critically at the heaviness of a Vanilla KUBERNETES deployment.

Standard (Vanilla) KUBERNETES (often referred to as K8s) is, in fact, engineered for large scale data centres. Its architecture strictly decouples the Control Plane from Worker nodes.

As briefly mentioned before, a standard Control Plane consists of many elements that contribute to the creation of a software bottleneck. Even in Single Node configuration (our case), where, of course, consensus is mathematically trivial (quorum is, of course, 1), `etcd` still performs extensive logging before committing any state change.

The baseline memory and CPU footprint of a Vanilla K8s deployment often exceeds our expectation and tends to create bottlenecks when deployed to mid-specs systems.

From K8s to K3s

To mitigate the architectural heaviness of a standard Vanilla KUBERNETES deployment, RANCHER LABS (now SUSE) developed K3S, a certified and lightweight KUBERNETES distribution designed specifically for resource-constrained and efficient cluster deployments.

K3S deploys Control Plane components as a single, cohesive entity, reducing memory overhead and avoiding unnecessary inter-process communication overhead.

Also, K3S optimizes on useless code, mainly dedicated to cloud providers deployments. K3S relies on KINE instead of `etcd`, which eliminates the need for consensus overhead on single-node setups.

Lastly, K3S remains compatible by default with many Container Network Interface (CNI), notably including MULTUS for fronthaul communication with specialized network equipments. We will shortly get in touch with MULTUS and CANAL CNIs.

As a result, baseline requirements for a K3S setup drops significantly and **do not sacrifice on many of the core elements** that made us choose a KUBERNETES environment in the first place [21].

2.3.4. Container Network Interfaces

The **Container Network Interface** (CNI) is a general specification defining how network interfaces are provisioned for Linux containers.

When a POD is scheduled, the KUBELET executes a CNI binary, passing networking configuration parameters. By default, KUBERNETES requires that a POD receives exactly one network interface (and address), generally referred to as `eth0`.

In our solution, we chose CANAL CNI.

Canal CNI

CANAL is not a standalone CNI: it is an **integration** of two landmarks in the realm of KUBERNETES CNIs: FLANNEL and CALICO.

FLANNEL mainly operates at routing and connectivity levels, ensuring proper encapsulation for intra-cluster communication.

When a packet needs to traverse from POD A running on Node 1 of the cluster to POD B running on Node 2 of the cluster, Flannel runs its backend:

- POD A sends a standard Ethernet frame to POD B's IP.
- The frame traverses virtual network structures and reached the Linux bridge on Node 1. We will briefly get in touch with Linux bridges shortly.
- The routing table stored in K3S data-store is used and POD B's subnet is mapped to Node 2's physical address.
- The Kernel encapsulates the original frame inside another one.

CALICO, instead, deploys a daemon on every node: it monitors the KUBERNETES API for policy entities. CALICO translates such logical policies in low-level kernel instructions, generally using Extended Berkley Packet Filters (eBPF).

Linux bridges

A Linux bridge is a **virtual device based on kernel software which operates by connecting two or more network interfaces** to form a single Local Area Network (LAN). It operates very similarly to a physical network switch: it operates at Layer 2 (data link layer) and stores a forwarding table.

In OKD, the default Linux bridge is **br-ex**: many telecommunication workflows are pre-configured to be compatible with **br-ex** by default, so we chose to migrate such behaviour to our implementation, mimicking the OPENSIFT one.

It will also be responsible for managing the INGRESS CONTROLLER and exposed services.

2.3.5. Multus CNI for 5G fronthaul

Many 5G deployments require precise timing constraints: the Precision Time Protocol (PTP) is the evolution of the ubiquitous Network Time Protocol (NTP) implemented via software in almost all operating systems, becoming the de facto standard for trivial clock synchronization.

PTP brings such concepts to a Local Area Network, where higher degrees of precision are possible, making it suitable for Mobile networks operations. PTP is often embedded in specialized network hardware, such as Small Form-factor Pluggable (SFP) transceivers-based network switches.

In KUBERNETES, MULTUS plays the role of the **intermediary for the fronthaul interface**: MULTUS is not a CNI in the traditional sense, it instead operates as a multiplexer [12].

When the KUBELET asks MULTUS to set-up a POD's network, it actually:

- Invokes the default CNI (CANAL in our case), which sets up `eth0` for traditional routing and assigns an IP from the default range.
- Parses the POD's annotations to detect the request for additional networking interfaces.
- For every additional interface requested, MULTUS invokes the related specialized CNI (SR-IOV, MACVLAN, ...).

An additional interface is specified via a `NetworkAttachmentDefinition`: in the following example, a powerful Software Defined Radio is adopted: a USRP N300, which we will extensively use when deploying our management platform.

The N300 USRP operates by using two SFP+ ports on a NIC: the first one acts as the primary one, the other (omitted here) is instead used only in case a supporting interface is needed.

```

1  apiVersion: "k8s.cni.cncf.io/v1"
2  kind: NetworkAttachmentDefinition
3  metadata:
4    name: oai-usrp-net
5  spec:
6    config: |
7      {
8        "cniVersion": "0.3.1",
9        "type": "macvlan",
10       "master": "enp65s0f1np1", # the physical network interface on our host
11       "mode": "bridge",
12       "mtu": 9000, # a higher requirement for the N300 USRP
13       "ipam": { "type": "static", "addresses": [
14         {
15           "address": "192.168.50.100/24", # IP that our pod will be
16             ↪ assigned on this interface
17           "gateway": "192.168.50.1", # optional
18         }
19       ] },
20     }

```

Interface definition in Ansible

The definition of **hardware characteristics, dependencies and modules** is a **pivotal element** of the ANSIBLE paradigm. Let's analyze and abstract from our PLAYBOOK, which shows the definition of Linux bridges and the double 10 GbE SFP+ networking interfaces, alongside persistent SCTP kernel module load and backing store creation for local PersistentVolumes in KUBERNETES.

```

1 - name: Gather current network configuration
2   ansible.builtin.setup:
3     gather_subset:
4       - network
5
6 - name: Generate configuration for bridge
7   ansible.builtin.template:
8     src: templates/bridge.yaml.j2
9     dest: /etc/netplan/60-k8s-bridge.yaml
10    mode: '0600'
11    # Security backup for eventual restore
12    backup: true
13
14 - name: Generate configuration for high-speed USRP N3xx interfaces
15   ansible.builtin.template:
16     src: templates/usrp.yaml.j2
17     dest: /etc/netplan/70-usrp.yaml
18     mode: '0600'
19     # Security backup for eventual restore
20     backup: true
21 ...
22 - name: Apply the new Netplan configuration
23   ansible.builtin.command: netplan apply
24   async: 45 # do not stop the playbook immediately, if the connection briefly
25             ↪ drops at this time
26   poll: 0
27 ...
28 - name: Make the SCTP load persistent across reboots
29   ansible.builtin.copy:
30     content: "sctp\n"
31     dest: /etc/modules-load.d/sctp.conf
32     mode: '0644'

```

```

32
33 - name: Create PersistentVolume local directory
34   ansible.builtin.file:
35     path: "/home/polimi/pv"
36     state: directory
37     owner: "{{ ansible_user }}"
38     group: "{{ ansible_user }}"
39     mode: '0777'

```

The interface definition, which uses `netplan`, stems from a template (with `j2` extension), an ANSIBLE object which merges a pre-defined artifact with default values and variables [19].

```

1 network:
2   version: 2
3   ethernets:
4     {{ br_interface }}:
5       dhcp4: false
6       dhcp6: false
7       # Important: disables IP connectivity on the original host interface
8       addresses: []
9   bridges:
10    {{ br_interface_name }}:
11      # Clones the MAC address from {{ br_interface }} on the bridge
12      macaddress: {{ ansible_default_ipv4.macaddress }}
13      interfaces: [{{ br_interface }}]
14      addresses:
15        # Automatically gathers the current IP and subnet mask (in CIDR
16        ↪ notation)
17        - {{ ansible_default_ipv4.address }}/{{ ansible_default_ipv4.prefix }}
18      nameservers:
19        addresses:
20          - {{ br_nameserver_1 }}
21          - {{ br_nameserver_2 }}
22          - {{ br_nameserver_3 }}
23        search:
24          - {{ br_search_domain }}
25      routes:
26        - to: "default"

```

```

26         via: {{ ansible_default_ipv4.gateway }}
27     parameters:
28         stp: false
29         forward-delay: 0

```

The N300 USRP SFP+ interfaces are similarly defined as follows:

```

1 network:
2     version: 2
3     ethernets:
4         {{ usrp_interface_1 }}:
5             addresses:
6                 - {{ usrp_interface_1_ip }}
7             mtu: 9000
8             dhcp4: false
9             dhcp6: false
10        {{ usrp_interface_2 }}:
11            addresses:
12                - {{ usrp_interface_2_ip }}
13            mtu: 9000
14            dhcp4: false
15            dhcp6: false

```

In which we exemplify the usage of variables and defaults in ANSIBLE:

```

1 ---
2 br_interface: eno1
3 br_interface_name: br-ex
4 br_nameserver_1: 1.1.1.1
5 br_nameserver_2: 8.8.8.8
6 br_nameserver_3: 8.8.4.4
7 br_search_domain: polimi.lab
8 usrp_interface_1: enp65s0f0np0
9 usrp_interface_2: enp65s0f1np1
10 usrp_interface_1_ip: 192.168.50.1/24
11 usrp_interface_2_ip: 192.168.60.1/24

```

2.3.6. A brief overview: SR-IOV and DPDK

Single Root I/O Virtualization (SR-IOV) is a PCIe standard that allows a single physical NIC (Physical Function) to appear as a multiple, independent virtual network cards (Virtual Functions), at the hardware level. Unlike Linux bridges, traffic flowing through a virtual function is switched directly by the circuitry on the NIC.

SR-IOV can be used in combination with MULTUS, moving virtual functions into the namespace of the POD.

To achieve pure determinism, 5G network functions can be run in combination with the Data Plane Development Kit (DPDK), whose main aim integrating the use of the Input/Output Memory Management Unit (IOMMU) to map memory buffers of the hardware virtual function directly into the user-space memory of the 5G network application.

2.3.7. High Availability and Virtual IPs

Having now completed both the management and deterministic data planes, the next natural requirement for a distributed 5G deployment is **High Availability** (HA).

In a multi-node cluster, relying on a single IP address of a specific server is a Single-Point-of-Failure (SPOF): when a node crashes, all services lose access to the KUBERNETES API and the Core network. The architectural solution is the implementation of a Virtual IP (VIP).

A Virtual IP is an IP address that is **not permanently tied to a specific physical NIC or MAC address**, but it is instead a floating address that can be dynamically claimed by any healthy node in a cluster.

To achieve this, the Address Resolution Protocol (ARP) kicks in: when a node crashes, another node broadcasts a **Gratuitous ARP reply**, an unsolicited ARP packet announcing that the Virtual IP address can now be reached through the other node's physical MAC address, instead of the first, initial one.

KEEPAIVED is a general and stable implementation of this concept, which is however unaware of containerized workloads and KUBERNETES, being it a general daemon for Virtual IP.

KUBE-VIP is instead running inside KUBERNETES, most commonly via a set of static PODS. Of course, it understands the health of KUBERNETES cluster and APIs, making it a perfect fit for our use-case.

The deployment of KUBE-VIP is in itself another example of the usage of the KUBERNETES API and CLI:

```

1 - name: Create directory for Kube VIP Manifest
2   ansible.builtin.file:
3     path: "/var/lib/rancher/rke2/server/manifests"
4     state: directory
5     mode: "0755"
6   when: inventory_hostname in groups['servers']
7
8 # Copy kube-vip to server 1 manifest folder for auto deployment at bootstrap
9 - name: Deploy Kube VIP Configuration
10  ansible.builtin.template:
11    src: templates/kube-vip-config.j2
12    dest: /var/lib/rancher/rke2/server/manifests/kube-vip.yaml
13    owner: root
14    group: root
15    mode: "0644"
16  when: inventory_hostname == groups['servers'][0]
```

In which we exemplify the usage of autonomous manifest deployment when bootstrapping KUBERNETES and also provide a typical framing for KUBERNETES manifests.

We also identify one of the most relevant details to our solution: **generality**.

Since any set of actions running within ANSIBLE can be generalized to be extended to potentially any group of machines, our solution is already valid for **scalable deployments**.

```

1 apiVersion: apps/v1
2 kind: DaemonSet
3 metadata:
4   creationTimestamp: null
5   labels:
6     app.kubernetes.io/name: kube-vip-ds
7     app.kubernetes.io/version: {{ kube_vip_version }}
8   name: kube-vip-ds
9   namespace: kube-system
10 spec:
11   selector:
12     matchLabels:
13       app.kubernetes.io/name: kube-vip-ds
```

```
14   template:
15     metadata:
16       creationTimestamp: null
17       labels:
18         app.kubernetes.io/name: kube-vip-ds
19         app.kubernetes.io/version: {{ kube_vip_version }}
20     spec:
21       affinity:
22         nodeAffinity:
23           requiredDuringSchedulingIgnoredDuringExecution:
24             nodeSelectorTerms:
25               - matchExpressions:
26                 - key: node-role.kubernetes.io/master
27                   operator: Exists
28               - matchExpressions:
29                 - key: node-role.kubernetes.io/control-plane
30                   operator: Exists
31     containers:
32     - args:
33       - manager
34       env:
35       - name: vip_arp
36         value: "true"
37       - name: port
38         value: "6443"
39       - name: vip_interface
40         value: {{ vip_interface }}
41       - name: vip_cidr
42         value: "32"
43       ...
44     image: ghcr.io/kube-vip/kube-vip:{{ kube_vip_version }}
45     imagePullPolicy: Always
46     name: kube-vip
47     resources: {}
48     securityContext:
49       capabilities:
50         add:
51         - NET_ADMIN
52         - NET_RAW
53     hostNetwork: true
```

```

54     serviceAccountName: kube-vip
55     tolerations:
56     - effect: NoSchedule
57       operator: Exists
58     - effect: NoExecute
59       operator: Exists
60     updateStrategy: {}
61     ...

```

2.4. System tuning for telecommunications workloads

When deploying software-defined 5G stacks (like OPENAIRINTERFACE and OPEN5GS, which we will shortly use extensively), the **primary challenge is determinism** [4].

General Purpose Processors are designed for throughput and fairness, using mechanisms like Symmetric Multi-processing (SMP) and power-saving states (C-states). As anticipated, optimizing for telecommunications and network workloads requires shifting some of the fundamental paradigms in systems development (though we should remember many optimizations are already there by default).

2.4.1. Boot-time Kernel optimizations

The foundation for determinism begins at the bootloader (in our case, Grand Unified Bootloader, GRUB). Passing specific parameters to the Linux Kernel via GRUB allows for **deep customization of hardware resource usage** before the OS scheduler can even manage them.

CPU Isolation and Thread Management

As per our laboratory workstation (12 physical cores, 24 logical ones, 64 GB system memory), the following configuration is applied:

```

isolcpus=2-11,14-23 nohz_full=2-11,14-23 rcu_nocbs=2-11,14-23
↪ kthread_cpus=0-1,12-13 irqaffinity=0-1,12-13

```

In which the partition is described as follows:

- **isolcpus**: Instructs the Kernel scheduler to avoid placing generic user-space processes on specified cores.

These cores are **exclusively reserved** for OAI gNBs and allows for precise energy consumption tracking.

- **nohz_full**: Enables a tickless kernel on the isolated cores.
Normally, Linux relies on a timer interrupt to manage scheduling and accounting. **nohz_full** prevents interrupt-driven latency spikes.
- **rcu_nocbs**: Read-Copy-Update (RCU) is a synchronization mechanism in the Linux Kernel. This parameter offloads RCU callbacks from the isolated cores to the house-keeping cores.
- **kthread_cpus** and **irqaffinity**: Force kernel threads and hardware interrupts to house-keeping cores (0-1, 12-13) in our case.

A note on symmetry

When Symmetric Multi-Threading (SMT) is enabled, a **single physical CPU core is shared across two logical threads**, separated by a predictable and constant offset, related, of course, to the number of physical CPU cores.

Physical core 0 maps, therefore, to threads 0 and 12, core 1 maps to threads 1 and 13, etc.

When planning displacement of scheduled PODS, an appropriate pair of threads must be defined, as they will also share the same physical execution pipeline and cache registers.

In Linux, `lscpu -e` confirms our analysis:

	CPU	NODE	SOCKET	CORE	L1d:L1i:L2:L3	ONLINE	MAXMHZ	MINMHZ	MHZ
1									
2	0	0	0	0	0:0:0:0	yes	5380.0000	545.0000	4962.1699
3	1	0	0	1	1:1:1:0	yes	5380.0000	545.0000	4962.9438
4	2	0	0	2	2:2:2:0	yes	5380.0000	545.0000	4962.0142
5	3	0	0	3	3:3:3:0	yes	5380.0000	545.0000	4937.1890
6	4	0	0	4	4:4:4:0	yes	5380.0000	545.0000	4970.2109
7	5	0	0	5	5:5:5:0	yes	5380.0000	545.0000	4947.4639
8	6	0	0	6	8:8:8:1	yes	5380.0000	545.0000	4974.1992
9	7	0	0	7	9:9:9:1	yes	5380.0000	545.0000	4973.5342
10	8	0	0	8	10:10:10:1	yes	5380.0000	545.0000	4972.6240
11	9	0	0	9	11:11:11:1	yes	5380.0000	545.0000	4964.5161
12	10	0	0	10	12:12:12:1	yes	5380.0000	545.0000	4978.2969
13	11	0	0	11	13:13:13:1	yes	5380.0000	545.0000	4976.7388
14	12	0	0	0	0:0:0:0	yes	5380.0000	545.0000	4963.7100

15	13	0	0	1 1:1:1:0	yes	5380.0000	545.0000	4955.4121
16	14	0	0	2 2:2:2:0	yes	5380.0000	545.0000	4954.6162
17	15	0	0	3 3:3:3:0	yes	5380.0000	545.0000	4951.6348
18	16	0	0	4 4:4:4:0	yes	5380.0000	545.0000	4952.5352
19	17	0	0	5 5:5:5:0	yes	5380.0000	545.0000	4960.9580
20	18	0	0	6 8:8:8:1	yes	5380.0000	545.0000	4955.6621
21	19	0	0	7 9:9:9:1	yes	5380.0000	545.0000	4952.1680
22	20	0	0	8 10:10:10:1	yes	5380.0000	545.0000	4952.2881
23	21	0	0	9 11:11:11:1	yes	5380.0000	545.0000	4952.3999
24	22	0	0	10 12:12:12:1	yes	5380.0000	545.0000	4955.5752
25	23	0	0	11 13:13:13:1	yes	5380.0000	545.0000	4955.8789

2.4.2. Memory management

```
default_hugepagesz=1G hugepagesz=1G hugepages=16 amd_iommu=on iommu=pt
```

- **Hugepages:** Standard memory pages have 4 KB size. A 1 GB memory allocation requires therefore roughly 260.000 pages, meaning the Translation Lookaside Buffer (TLB) will frequently miss, requiring page table walks.

Pre-allocating 1 GB HUGEPAGES minimizes TLB misses, which is crucial for the size of memory buffers being used in 5G network processing.

- **iommu=pt:** Enables pass-through mode for the IOMMU. This allows PCIe devices (like specialized network equipment) to map memory directly into user-space applications, via Virtual Function I/O.

This is a common prerequisite for SR-IOV and DPDK.

2.4.3. Run-time operating system tuning

While GRUB configures the system at boot-time, the `tuned` daemon, as anticipated, is responsible for applying dynamic run-time configurations.

Tuning requires adapting `/etc/tuned/*/tuned.conf` with relevant parameters.

We will mainly focus on the most important assets to us: **network latency and virtual memory management**.

CPU governor and C-states

```
2 force_latency=1
3 governor=performance
4 energy_perf_bias=performance
5 min_perf_pct=100
```

Sets up high base parameters for 5G network workloads and allows for deterministic performance metrics and latency, by minimizing run-time switches.

Virtual memory tuning

```
1 [vm]
2 dirty_ratio=10
3 dirty_background_ratio=3
4 swappiness=10
5 transparent_hugepages=never
```

- **Dirty ratios:** these parameters governs cached data writes to disk. Lowering them prevents massive I/O bottlenecks. When too much memory is **dirty**, the Kernel block processes to flush data: lower ratios force frequent, smaller flushes.
- **Swappiness:** lowering the **swappiness** parameter ensures the kernel keeps application memory in RAM rather than paging to disk. When deploying a KUBERNETES cluster, SWAP is often disabled entirely.
- **Transparent Hugepages:** disables dynamic building of HUGE_PAGES. Since explicit 1 GB HUGE_PAGES are already configured in GRUB, having this enabled would result in unpredictable latency spikes.

2.4.4. Network stack tuning

```
1 [sysctl]
2 net.core.busy_poll=50
3 net.core.busy_read=50
4 net.ipv4.tcp_fastopen=3
5
6 fs.inotify.max_user_instances=256
7
8 kernel.sched_rt_runtime_us=1000000
9
```

```

10 net.core.rmem_default=134217728
11 net.core.rmem_max=134217728
12 net.core.wmem_default=134217728
13 net.core.wmem_max=134217728
14
15 [net]
16 devices=enp65s0f0np0,enp65s0f1np1
17 ring_rx=4096
18 ring_tx=4096

```

Ring buffers

A Ring Buffer is a circular queue of fixed size implemented in memory shared between the NIC and the operating system driver via Direct Memory Access (DMA). There are two independent rings per network queue: the RX and TX ring.

The elements within these rings are descriptors: small data structures containing physical memory pointers to an allocated block of host RAM, where the actual packet payload can be written.

The typical suggested standards for 5G deployments are:

```

sudo ethtool -G enp65s0f0np0 rx 4096 tx 4096
sudo ethtool -G enp65s0f1np1 rx 4096 tx 4096

```

With `enp65s0f0np0` and `enp65s0f1np1` being the interface names for SFP-terminated network plugs.

The default ring buffer size on standard Linux distributions is typically set to 256 or 512 descriptors to minimize memory footprint of idle interfaces.

To provide a reasonable estimation for a typical 5G deployment, let's compute the time to exhaust a 512-descriptor ring under a 10 Gb/s fronthaul link processing standard Ethernet frames (assuming average packet size $P = 1200$ bytes).

The packet arrival rate R_p is defined as:

$$R_p = \frac{\text{Throughput}}{\text{Packet size in bits}} = \frac{10 \cdot 10^9}{1200 \cdot 8} \simeq 1041666 \text{ packets per second}$$

The time window T_e available to the CPU to respond to an interrupt and completely drain a 512-descriptor ring before a hardware drop occurs is:

$$T_e = \frac{\text{Ring size}}{R_p} = \frac{512}{1041666} = 491,52 \mu s$$

A window of 491,52 μs is quite narrow. If the kernel enters a non-preemptible section or a cache miss stalls the handler for such a period, as anticipated, a packet drop is guaranteed.

Expanding the ring to 4096 descriptors allows the time window to expand linearly:

$$T_e = \frac{\text{Ring size}}{R_p} = \frac{4096}{1041666} = 3,93 ms$$

A 3,93 ms buffer provides a large safety margin, allowing the real-time scheduler to context-switch into the relevant thread and drain hardware queues accordingly.

Similarly, the maximum capacity for the software ingress queue and synchronous polling are provided with a higher threshold.

Virtual File-system tuning for containerization

The `inotify` subsystem is a Kernel feature that allows applications to monitor file-system events (e.g., file creation, modification, deletion) natively without polling loops.

In a KUBERNETES environment, `inotify` is heavily used by:

- The container runtime (`containerd`): to track changes in container roots, namespaces and standard image runtime configuration.
- KUBELET: to monitor configuration maps, mounts and container log paths.
- CNI plugins: to monitor changes in network configuration files and inter-process communication.

The `max_user_instances` parameter defines the upper limit on the number of independent `inotify` monitoring instances a single user can instantiate simultaneously across the system.

In our implementation, we quickly realized that the default value of 128 is not sufficient. The typical suggested value for a 5G stack is in fact 256 or 512.

2.4.5. Kubernetes resource management

We now need to port this knowledge to KUBERNETES for resource management.

```

1 kind: KubeletConfiguration
2 apiVersion: kubelet.config.k8s.io/v1beta1
3
```

```

4  featureGates:
5    CPUManagerPolicyOptions: true
6    CPUManagerPolicyAlphaOptions: true
7
8  cpuManagerPolicy: static
9  cpuManagerPolicyOptions:
10    strict-cpu-reservation: "true"
11    full-pcpus-only: "true"
12
13 reservedSystemCPUs: "0-1,12-13"
14 topologyManagerPolicy: "single-numa-node"

```

CPU management policies

```
cpuManagerPolicy: static
```

By default, KUBERNETES uses a `none` policy, meaning PODS share a common pool of CPUs.

The `static` policy fundamentally alters this behaviour: when a POD requests an integer number of CPUs and is placed in the GUARANTEED Quality of Service (QoS) class, the KUBELET uses Kernel abstractions to bind that POD to specific, physical CPU cores. No other PODS are allowed on those cores, guaranteeing no performance interference.

```
reservedSystemCPUs: "0-1,12-13"
```

This setting explicitly instructs KUBERNETES that cores 0, 1, 12 and 13 belong exclusively to the host operating system and KUBERNETES internal daemons (housekeeping). This perfectly aligns with `isolcpus` as seen earlier.

```

1  cpuManagerPolicyOptions:
2    strict-cpu-reservation: "true"
3    full-pcpus-only: "true"

```

When enabled, these options forces the KUBELET to validate that reserved CPU cores match physical resource allocations.

As anticipated, in architectures supporting Symmetric Multi-threading (SMT) a single physical core presents itself as two logical threads.

`full-pcpus-only` imposes PODS to request only even number of cores, to make all PODS

in GUARANTEED QoS receive symmetric CPU threads.

3 | A Neutral Host management platform

3.1. A primer on Kubernetes orchestration

In a KUBERNETES environment, the desired state of the cluster and its deployments are described using Yet Another Markup Language (YAML) manifests, which can be employed in a self-contained static setting or a packaged strategy.

3.1.1. Static declarative manifests

The most fundamental way of deploying an application to KUBERNETES is passing a **static YAML manifest** to the KUBERNETES CLI (`kubectl`).

The following example provides the Access and Management Function (AMF) for OPEN5GS, an open-source implementation of the 5G Core.

We will shortly use OPEN5GS in our platform.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: open5gs-amf
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: open5gs-amf
10   template:
11     metadata:
12       labels:
13         app: open5gs-amf
14   spec:
```

```
15     containers:
16       - name: amf
17         image: gradient/open5gs:2.7.7
18         env:
19           - name: MCC
20             value: "999"
21           - name: MNC
22             value: "91"
```

Executing `kubectl apply -f open5gs-amf.yaml` results in the following actions being executed:

- The `kubectl` client parses the YAML file, converts it to a serializable format and uses the KUBERNETES API to check for the presence of a resource matching the same deployment name.
- If the specified resource does not exist, it is simply created.
- If, instead, a matching resource with that name exists, a **merge** procedure is applied, patching the current state in `etcd` with the new static manifests and annotating the `last-applied-configuration` parameter.

Of course, even if static manifests are **idempotent and declarative, they are not a canonical implementation of the Infrastructure as Code paradigm**. A 5G Core deployment requires much more services than the AMF (roughly 10-15) and each of them requires its own `Deployment`, `Service`, `ConfigMap` and networking configuration.

While they are **ideal for smaller projects**, in this case, the adoption of static manifests results in a scattered solution consisting of tens of YAML files. Configuring a new instance requires altering each of them manually, which is inherently against IaC principles and inevitably leads to configuration drifts.

3.1.2. The Helm package manager

To resolve the rigidity of static manifests, the HELM package manager introduces the concept of **templating** to KUBERNETES.

A HELM package is often referred to as `CHART`, consisting of a structured directory containing GO text templates, a `values.yaml` file for variable injection and metadata.

HELM is a purely client-side architecture, a `CHART` is simply deployed by:

```
helm install <name> <path> -n <namespace>
```

With the name being the chosen deployment name. Its value is arbitrarily assigned, and it should be coherent with values specified inside the CHART, whose main directory is the relative (or absolute) path specified.

A **namespace** in Kubernetes is a logical partition within the cluster, designed to isolate groups of resources.

For example:

```
helm install open5gs charts/open5gs -n core
```

After the `install` command is given, the HELM binary processes GO templates, injecting variables and compiling the final manifests in memory. As the KUBERNETES CLI, HELM connects directly to the KUBERNETES API.

The state of the resulting deployment is stored as a **RELEASE** directly inside the chosen **namespace**. This allows for agile rollback and upgrade management.

Templating

As anticipated, one of the advantages of using HELM in our solution is its templating engine. Let's, for example, rewrite our previous static manifest as a small HELM CHART.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: {{ .Release .Name }}-amf
5 spec:
6   replicas: {{ .Values.amf.replicaCount }}
7   selector:
8     matchLabels:
9       app: open5gs-amf
10  template:
11    metadata:
12      labels:
13        app: open5gs-amf
14    spec:
15      containers:
```

```

16     - name: amf
17       image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
18       env:
19         - name: MCC
20           value: "{{ .Values.global.mcc }}"
21         - name: MNC
22           value: "{{ .Values.global.mnc }}"

```

With a matching `values.yaml` being the following:

```

1 global:
2   mcc: "999"
3   mnc: "91"
4 amf:
5   replicaCount: 1
6 image:
7   repository: gradient/open5gs
8   tag: 2.7.7

```

Such flexibility will allow us to replicate our base OPENAIRINTERFACE GNB configuration and extend it to support additional tenants.

3.2. The Open-Source 5G Ecosystem

The democratization of 5G deployment allowed for the development of comprehensive open-source software stacks [23]. The three most prominent players in this space are OPEN5GS, SRSRAN and OPENAIRINTERFACE (OAI).

3.2.1. A quick overview at srsRAN

SRSRAN, developed by Software Radio Systems is a software radio suite capable of providing both User Equipment (UE), eNodeB (4G Base Station) and gNodeB (5G Base Station).

SRSRAN is written almost entirely in C and C++, targeting C11 features. Its defining characteristics are its **portability and ease of use**.

It is designed to be highly readable, modular and easy to deploy on generic Software Defined Radios (SDRs) like ETTUS RESEARCH's USRPs.

`srsRAN_Project` focuses exclusively on the 5G CU/DU. It heavily leverages DPDK for

packet processing and is designed to interface with O-RAN compliant 7.2x Radio Units (RUs). It generally couples with third-party solutions (like OPEN5GS) to provide the Core Network.

During our testing, we found out that SRSRAN-based tenants **tend to consume much more resources** than OAI-based ones in equal operating conditions.

Logging strategies and compatibility are also not ideal for our solution, which led to the ultimate decision of shifting our focus towards OAI.

3.2.2. Open5GS

OPEN5GS is an open-source C implementation of the 5G Core and 4G Evolved Packet Core (EPC). During its development, it has evolved into one of the most stable, production-ready and widely adopted open-source Core Networks in the industry.

Control Plane functions are **decoupled microservices**: they communicate across an internal message bus, the SERVICE-BASED INTERFACE (SBI) using HTTP with JSON payloads.

A full deployment of OPEN5GS via HELM instantiates the following critical components as independent PODS:

- **Network Repository Function (NRF)**: the central DNS/Service Mesh. When the AMF boots up, it registers its IP and capabilities with the NRF. When the SMF needs to communicate with the AMF, it queries the NRF.
- **Access and Management Function (AMF)**: as previously mentioned, the single entry point for the GNB and the UE.
It terminates the N2 interface (NGAP protocol) from the radio and manages UE registration, authentication and mobility (often referred to as handover).
- **Session Management Function (SMF)**: manages the establishment, modification and release of the Protocol Data Unit (PDU). It allocates IP addresses to the UEs.
- **User Plane Function (UPF)**: is responsible for routing the actual internet traffic generated by the UE.
It terminates the N3 interface (GTP-U tunnels) from the GNB and routes packets to the Data Network.
- **Unified Data Management (UDM) and Authentication Server Function (AUSF)**: Replaces the Home Subscriber Service (HSS). It stores subscriber data,

cryptographic keys and generates authentication challenges.

- **Policy Control Function (PCF)**: manages QoS policies and billing rules.
- **MongoDB**: all subscriber data, session state and persistent configuration details are stored in a MONGODB backend.

Thanks to the separation of Control and User Planes, deployment of the Control Plane can be entirely separated (e.g. in the cloud) while multiple PODS can live in edge servers, located directly inside meaningful facilities, creating an advantage for data locality and latency.

3.2.3. OpenAirInterface

The OPENAIRINTERFACE Software Alliance (Eurecom) provides a complete, 3GPP-compliant software stack, which includes the UE, RAN and Core. We will make extensive use of the OAI gNB in our solution.

OAI respects the 3GPP **functional splits** we described at the beginning: the Centralized Unit and Distributed Unit.

The CU handles the higher layers of the protocol stack, which are not strictly time-sensitive: it is typically split in CU-CP (Radio Resource Control, terminates the N2 interface to the AMF) and CU-UP (handles user data and terminates the N3 interface to the UPF).

The DU handles the highly deterministic, real-time layers: RLC, MAC and Physical resources. It communicates with the CU via the F1 interface.

The cloud-native paradigm can provide additional malleability by formalizing a scattered implementation strategy across regional data centres and edge deployments.

The exceptional **logging and RAN Intelligent Controller integration** capabilities of OAI made it the ideal choice for our solution.

3.2.4. FlexRIC

The OPENAIRINTERFACE Software Alliance also developed FLEXRIC, a highly modular, performance-oriented implementation of the Near-Real-Time RAN Intelligent Controller (Near-RT RIC).

In the paradigm of Cloud-Native 5G, FLEXRIC acts as the **Software Defined Networking equivalent for the RAN**, physically extracting Radio Resource Management metrics and making them available into a fully programmable entity.

Architecturally, the system is divided into two primary components: the E2 AGENT and the RIC Control plane.

The core routing and encoding engines of FLEXRIC are written in C, while XAPPS (independent software microservices running on FLEXRIC) can be written in both C and PYTHON.

XAPPS use standardized E2 Service Models (E2SM):

- **E2SM-RC** (RAN Control): these XAPPS can asynchronously inject policy decisions back into the RAN (for example, scheduling algorithms for prioritization).
- **E2SM-KPM** (Key Performance Measurements): these XAPPS can dynamically subscribe to sub-millisecond telemetry streams, including MAC and Physical layer buffer status or metrics.

While both service models are perfectly aligned with our use case, **we will choose KPM**, since it will allow us a higher degree of flexibility.

3.3. Deploying the Core Network

As anticipated, OPEN5GS is the ideal fit for our solution. Its deployment will require spinning up ~ 10 stateless microservices implementing the aforementioned Network Functions, carrying User and Control planes.

GRADIANT provides a comprehensive set of cloud-native tools to deploy, automate and orchestrate OPEN5GS. Among those, a HELM CHART template serves as the foundation for our deployment.

3.3.1. Structural overview

Chart.yaml reveals the dependency structure:

```

1  apiVersion: v2
2  appVersion: "2.7.2"
3  description: >
4    Helm chart to deploy Open5gs services on Kubernetes.
5  home: https://github.com/gradiant/5g-charts
6  ...
7  dependencies:
8    - name: common

```

```
9     repository: oci://registry-1.docker.io/bitnamicharts
10     tags:
11         - bitnami-common
12     version: 1.x.x
13 - name: mongodb
14     version: ~12.1.19
15     repository: file://../mongodb
16     condition: mongodb.enabled
17 - name: open5gs-amf
18     version: ~2.2.5
19     repository: file://../open5gs-amf
20     condition: amf.enabled
21     alias: amf
22     ...
23 - name: open5gs-smf
24     version: ~2.2.6
25     repository: file://../open5gs-smf
26     condition: smf.enabled
27     alias: smf
28     ...
29 - name: open5gs-upf
30     version: ~2.2.6
31     repository: file://../open5gs-upf
32     condition: upf.enabled
33     alias: upf
34 - name: open5gs-webui
35     version: ~2.2.5
36     repository: file://../open5gs-webui
37     condition: webui.enabled
38     alias: webui
```

In which we immediately identify the main components of our 5G Core: we will extensively interact with the AMF, SMF and UPF, while we will shortly make use of the management WEBUI.

We also highlight the presence of **condition variables**, which allow for a purely distributed deployment (e.g. keeping the Control Plane in a set of centralized nodes while moving the User Plane closer to the edge).

3.3.2. Authentication and key management

In 5G, Authentication relies on the Authentication and Key Agreement (AKA) algorithm, in which symmetric cryptographic keys are employed [15].

From the standpoint of the final user, cryptographic secrets are physically burned into the physical SIM card (or eSIM).

The two primary symmetric keys are:

- **Ki (Subscriber Secret Key)**: a 128/256-bit root secret identifying the subscriber.
- **OPc (Operator Variant Algorithm Configuration Field)**: a derived key mathematically generated from Ki and an additional Operator Key (OP).

In our deployment, we pre-populate the database via a specific population job, which injects them at bootstrapping time.

For each UE, we will need to define its International Mobile Subscriber Identity (IMSI), Access Point Name (APN), Ki, OPC. Remaining parameters will be inferred and completed by default. We can also alter them after the insertion, by logging into the Web UI.

```

1 dbURI: "mongodb://{ .Release.Name }-mongodb/open5gs"
2
3 populate:
4   enabled: true
5   image:
6     registry: docker.io
7     repository: gradient/open5gs-dbctl
8     tag: 0.10.3
9     pullPolicy: IfNotPresent
10  initCommands:
11    - open5gs-dbctl add_ue_with_apn 999910000000001
12      ↪ 3114A38F43110BFD775B8D97F8493CAB 7E9CF00650C37664C73FC7391B9BB4DB
13      ↪ tenant1
14    - open5gs-dbctl add_ue_with_apn 999920000000001
15      ↪ DA40AA0E1663C9B6E8E0B7DD1775A5D4 53407D8972223590280DF61CA9A9FBCD
16      ↪ tenant2
17    - open5gs-dbctl add_ue_with_apn 999920000000002
18      ↪ EFFA30163D68A796D7BDF40F5589113D B135331676A01E4BBD17EC3B894A9C79
19      ↪ tenant2

```

We inject 3 UEs with PLMNs 99991 and 99992: **PLMNs having 999** as their Mobile Country Code (MCC) are **dedicated to test/internal-use networks**, so they are perfectly aligned to our use case.

The first parameter following the IMSI is the KI (e.g. 3114A38F43110BFD775B8D97F8493CAB), the following is the OPC (e.g. 7E9CF00650C37664C73FC7391B9BB4DB) and the last one is the APN (e.g. `tenant1`).

Programmable SIM cards and commercial UEs: a brief overview

SYSMOCOM programmable Subscriber Identity Modules (SIM) cards are specialized, programmable smart cards designed for telecommunications research, laboratory testing and private cellular networks.

Unlike standard commercial SIM cards, which lock network credentials permanently during factory manufacturing, programmable SIM cards **feature open and flexible encryption layers**. Depending on the specific model, they give the user complete or partial control on cryptographic secrets and identifiers.

Throughout our work, we used SYSMOISIM-SJA2 programmable SIM cards, which come pre-loaded with an IMSI from PLMN 99970 and attached cryptographic keys.



Figure 3.1: SYSMOCOM SYSMOISIM-SJA2 programmable SIM cards

They are also normally **paired with a Smart Card Reader/Writer** (in our case, a HID OMNIKEY 6121): SIM cards use the same underlying features as common smart cards.



Figure 3.2: HID OMNIKEY 6121 Smart Card Reader/Writer

GNU/Linux distributions include support for smart cards through PC/SC, a daemon that enables communications with readers and cryptographic tokens.

The Open Source Mobile Communication Project (OSMOCOM) includes a variety of software and tools implementing many communication standards. Among those, we will use PYSIM, a suite of PYTHON scripts for interfacing with both generic and programmable cards, supporting multiple interfaces, including PC/SC.

Writing to a fully programmable SIM card is as easy as **invoking the pySim-prog script with related parameters**, which crucially include the ADM1 PIN, **an administration secret provided by the issuer** of the SIM card, which cannot be over-written.

```
pySim-prog.py -p 0 --type=sysmoISIM-SJA2 --pin-adm=12345678 --mcc=999 --mnc=91
→ --imsi=999910000000001 --ki=3114A38F43110BFD775B8D97F8493CAB
→ --opc=7E9CF00650C37664C73FC7391B9BB4DB
```

With `-p 0` indicating the first PC/SC-compatible Smart Card writer on the system.

TECHSHIP and SIERRA WIRELESS are prominent companies in the embedded wireless hardware industry, operating as a distributor and a manufacturer, respectively.

Throughout our work, we used TECHSHIP's MC201, a specialized M.2 B-Key to USB-C adapter board, featuring dual NANO-SIM slots and $4 \times$ SMA antenna connectors. On TECHSHIP's adapter board, we mounted SIERRA WIRELESS EM9191, an industrial-grade 5G M.2 Module designed to provide high-speed cellular connectivity to IoT and enterprise applications, through QUALCOMM's chipset.

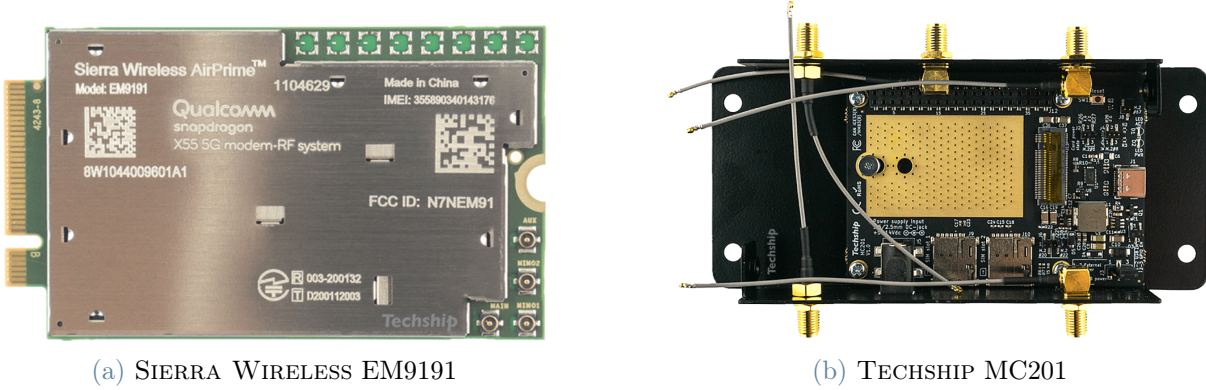


Figure 3.3: A commercial USB 3.0 setup for UE testing

We also made occasional use of commercial OPPO smartphones for testing [36].

3.3.3. Interacting with Sierra Wireless UEs

AT (Attention) commands are the standard **text-based interface for low-level modem configuration**, debugging and testing [38]. In Linux, the Kernel enumerates several character devices. Typically, `/dev/ttyUSB*` (e.g. `/dev/ttyUSB0`) serves as the primary AT command port.

Tools like `picocom` or `pyserial` allow users to **directly interact with the modem**, either through the CLI or via automation scripts.

For our use case, useful AT commands generally include:

- ATE1: enables local character echo.
- AT+CIMI: requests the IMSI from the inserted SIM card.
- AT!ENTERCND="A710": enters the privileged command mode, allowing for advanced NVRAM parameter tuning.
- AT!USBCOMP=1,6,109: in privileged command mode, enables QMI mode.
- AT!GSTATUS?: returns a detailed breakdown of the current radio state and metrics.
- AT+CGDCONT=1,"IP","tenant1": sets the primary APN to `tenant1`.
- AT!SELRAT=20: locks the modem into 5G only mode.
- AT+CFUN=X: turns the modem ON ($X = 1$) or OFF ($X = 0$).
- AT!RESET: restarts the modem.

Alongside SIERRA WIRELESS software development kit (MBPL SDK), which by default uses the Mobile Broadband Interface Model (MBIM) protocol, the transition to **Qualcomm MSM Interface (QMI)** allows us additional flexibility.

For example,

```
sudo systemctl disable ModemManager.service
qmcli -d "/dev/cdc-wdm0" --wds-start-network="apn='tenant1',ip-type=4"
→ --client-no-release-cid
udhcpc -i "wwan0"
```

Connects the first connected modem to the first tenant with APN `tenant1`, asking then for an IP address via DHCP on the exposed modem interface. If the GNU/Linux distribution of choice comes pre-loaded with `ModemManager`, if enabled, may conflict with QMI, so it should be disabled before continuing.

After completion, `ping -I wwan0 10.45.1.1` allows us to check for connectivity and `iperf3 -bind-dev wwan0 -R -t 0 -b 30M -c 10.45.1.1 -p 5201` runs an indefinite downlink speed test against the UPF, via TCP, with a target throughput of 30 Mb/s using `iperf`, an open-source command-line tool to measure network bandwidth and throughput.

Building up on such knowledge, **a comprehensive PYTHON management script is provided as a support** to our solution.

```
usage: modem_manager.py [-h] {info,start,reset,iperf} ...
```

Sierra 5G Modem Manager (IMSI/Tenant Routing & Diagnostics)

positional arguments:

{info,start,reset,iperf}

info	Scan modems and show IMSI, APN, and RF Telemetry
start	Auto-detect IMSIs, configure Raw IP, and connect to Tenants
reset	Send AT!RESET to all detected Sierra 5G modems
iperf	Run a continuous iperf3 test dynamically bound to a specific IMSI

For example, `sudo modem_manager.py iperf 999910000000001 30M tcp down 5201` identifies the SIERRA WIRELESS modem having IMSI 999910000000001 and spawns an `iperf` downlink test using TCP with a target throughput of 30 Mb/s.

3.3.4. Network slicing

A Neutral Host requires data isolation between different operators sharing a common infrastructure. This is achieved through network slicing, by providing Network Slice Assistance Information and Data Network Name (DNN) mappings.

This logic is enabled within the `smf` and `upf` blocks.

```

1  smf:
2      enabled: true
3      config:
4          ...
5      subnetList:
6          - subnet: 10.45.1.0/24
7            gateway: 10.45.1.1
8            dnn: tenant1
9          - subnet: 10.45.2.0/24
10           gateway: 10.45.2.1
11           dnn: tenant2
12          - subnet: 10.45.3.0/24
13           gateway: 10.45.3.1
14           dnn: tenant3
15          - subnet: 10.45.4.0/24
16           gateway: 10.45.4.1
17           dnn: tenant4
18      ...
19      info:
20          - s_nssai:
21              - sst: 1
22                dnn: [tenant1, tenant2, tenant3]

```

When UE having IMSI 999910000000001 attaches to the network, the SMF intercepts the PDU Session Establishment request. The DNN matches the APN and the SMF assigns an IP address from the 10.45.1.0/24 block.

A UE having IMSI 999920000000001 will instead receive an IP address from the 10.45.2.0/24 block, and so on.

3.4. Traffic policy enforcement

The most advanced architectural variation in our Core Network deployment is the use of a **custom-built container image for the User Plane Function (UPF)** which embeds two pivotal components to our solution: **iperf** for network validation and a **TRAFFIC CONTROLLER** to enforce custom limitation policies on specific UEs and sessions.

The UPF is responsible for **routing and forwarding user data**. It acts as the gateway connecting the RAN to the external Data Networks via the N6 interface. The UPF encapsulates user traffic using a tunnelling protocol.

As per [8] and [7], OPENAIRINTERFACE implementation of **pure radio slicing** (PRB blocklisting and Service Differentiation) is in active development, which restricts our abilities on traffic control and usage limitation [5].

We would therefore like to exploit Linux-native mechanisms to enforce network policies on UEs.

OPEN5GS materializes the N6 interface inside the Linux Kernel via a **virtual tunnelling device**, conventionally named **ogstun**. All traffic originating from the UEs heading towards external networks passes through this virtual interface.

Applying traffic control in Linux exploits directly the **ogstun** interface, and we would like to **act on it as a response to throttling requests**. Doing this will effectively **control the data rate of the UEs**.

3.4.1. Linux Traffic Control

The core mechanism to restrict bandwidth usage is the Linux **TRAFFIC CONTROL** subsystem. It governs how packets are queued, scheduled and dropped by networking interfaces.

Every network interface in Linux has an associated Queueing Discipline (**qdisc**). We will use the one known as **HIERARCHICAL TOKEN BUCKET (HTB)**.

HTB allows for the creation of a tree-like structure of classes, where each class can be assigned a guaranteed bandwidth and a maximum burst size.

HTB operates on the principle of a token bucket, in which tokens are generated at a constant rate (the defined bandwidth limit). To transmit a byte of data, **a token must be consumed from the bucket**. If the bucket is empty, the packet is **queued until enough tokens are generated**. The result is a strict enforcement of an upper limit on throughput [30].

Bootstrapping

The first step is the bootstrapping of the HTB environment on `ogstun`, using the `startup` event hook.

```

1 @app.on_event("startup")
2 def setup_tc_root():
3     subprocess.run(f"tc qdisc del dev {INTERFACE} root 2>/dev/null",
4         ↪ shell=True)
5     res_qdisc = subprocess.run(
6         ↪ f"tc qdisc add dev {INTERFACE} root handle 1: htb default 10", ...
7     )
8     res_class = subprocess.run(
9         ↪ f"tc class add dev {INTERFACE} parent 1: classid 1:10 htb rate 10000mbit",
10        ↪ ...
11    )

```

- Cleanup: attempts to delete any existing root `qdisc` on `ogstun`.
- Root creation: creates the root HTB `qdisc` with the handle 1. It sets the default handle to 10, which is defined in the following.
- Default class creation: Class 10 is created, with a rate of 10 Gb/s. This acts as the base pool: common UEs not being targeted by throttling will hit this class and experience full speed.

Class identifiers

In Traffic Control, every class needs a unique hexadecimal `classid` (an identifier) and every filter needs a priority (`prio`). To achieve a stateless approach which does not require a storage strategy for class identifiers, a **hashing-inspired algorithm** based on the user's IPv4 address is employed:

```

1 def get_class_and_prio(ip: str):
2     parts = ip.split(".")
3     octet3 = int(parts[2])
4     octet4 = int(parts[3])
5
6     minor_hex = f"{octet3:02x}{octet4:02x}"
7     prio_int = int(minor_hex, 16) # Hex to dec

```

```

8
9     return f"1:{minor_hex}", prio_int

```

For example, when providing 10.45.1.20, the controller isolates the last two octets (1 and 20), it converts them to 2-digits hexadecimal strings: 1 to 01 and 20 to 14, then concatenates them and converts them back to a base 10 integer for the priority filter.

Live bandwidth measurement

Before applying a percentage-based reduction, the controller must **determine the user's current baseline throughput**. This is achieved via a measurement strategy adopting tcpdump:

```

1 def measure_current_bandwidth(ip: str) -> float:
2     ...
3     cmd =
4     ↪ f"timeout {MEASUREMENT_TIME} tcpdump -i {INTERFACE} -n 'dst host {ip}'
5     ↪ -w - 2>/dev/null | wc -c"
6
7     try:
8         result = subprocess.run(cmd, shell=True, capture_output=True,
9         ↪ text=True)
10        bytes_captured = int(result.stdout.strip())
11        bytes_captured = max(0, bytes_captured - TCPDUMP_HEADER_SIZE)
12
13        # Simple Mb/s conversion
14        bits = bytes_captured * 8
15        mbps = bits / (MEASUREMENT_TIME * 1E6)
16        return mbps
17
18    ...

```

The result will be an average estimation of the throughput observed on the specified UE's IPv4 address in the last 15 seconds.

Packet classification and bandwidth enforcement

Once the target bandwidth is calculated, we then provision the dynamic HTB class and associated packet filter.

```

1  async def limit_lifecycle(ip: str, target_mbps: float, duration_minutes: int):
2      class_id, prio = get_class_and_prio(ip)
3      # Conflicting rules are deleted
4      remove_tc_rule(ip, class_id, prio)
5
6      rate_str = f"{max(MIN_RATE, int(target_mbps))}mbit"
7
8      subprocess.run(
9          f"tc class add dev {INTERFACE} parent 1: classid {class_id}
10             htb rate {rate_str} burst 100k quantum 60000",
11          shell=True,
12      )
13      subprocess.run(
14          f"tc filter add dev {INTERFACE} protocol ip parent 1:0 prio {prio}
15             flower dst_ip {ip} flowid {class_id}",
16          shell=True,
17      )

```

Because of segmentation, bursts of up to 60-100 KB are possible, so we should allow them even in case of an insufficient token provisioning and corresponding rate.

We also use the modern FLOWER classified, which leverages the Kernel's native flow dissector, allowing for clean matching rules and directs matching packets precisely into the newly created HTB class.

Orchestration and containerization

Most of the orchestration is managed through PYTHON FASTAPI's asynchronous primitives, which **expose the POST /throttle** through HTTP on port 8000.

To adapt the OPEN5GS UPF container image accordingly, we will need to ensure it includes required dependencies:

```

1  FROM radiant/open5gs:2.7.5
2
3  # Temporarily switch to the root user to allow for dependencies installation
4  USER root
5
6  RUN ... && /usr/bin/apt-get -y install --no-install-recommends python3
   ↪ python3-fastapi python3-uvicorn python3-pydantic ...

```

```

7
8 ##### COPY AND PREPARE THE CONTROLLER SCRIPT #####
9 RUN /bin/mkdir -p /controller/
10 COPY ./controller.py /controller/
11
12 USER open5gs

```

And expose a KUBERNETES **service** on our cluster:

```

1 apiVersion: v1
2 kind: Service
3 ...
4 spec:
5     type: ClusterIP
6     selector:
7         app.kubernetes.io/name: upf
8         app.kubernetes.io/instance: open5gs
9     ports:
10         - port: 8000
11           targetPort: 8000

```

Finally, our controller is invoked through a custom launch command on the UPF container, alongside iperf3 servers **in the background**:

```

1 containers:
2   - name: {{ include "common.names.fullname" . }}
3     image: {{ template "open5gs.upf.image" . }}
4     imagePullPolicy: {{ .Values.image.pullPolicy | quote }}
5     command:
6       - /bin/sh
7       - -c
8       - |
9         ...
10        # Now, run the background services
11        echo "Starting iperf3 servers and controller in the background..."
12        stdbuf -o0 iperf3 -s &
13        stdbuf -o0 iperf3 -s -p 5202 &
14        stdbuf -o0 iperf3 -s -p 5203 &
15        stdbuf -o0 iperf3 -s -p 5204 &

```

```

16  stdbuf -o0 iperf3 -s -p 5205 &
17  /usr/bin/stdbuf -o0 /bin/bash -c 'cd /controller && /usr/bin/python3 -m
    ↪  uvicorn controller:app --host 0.0.0.0 --port 8000' &
18
19  # Finally, run the main application in the foreground
20  # This is what keeps the container running.
21  echo "Starting Open5GS UPF..."
22  open5gs-upfd

```

The resulting container image can be set-up so to update periodically via automated repository actions (e.g. GITHUB WORKFLOWS ACTIONS).

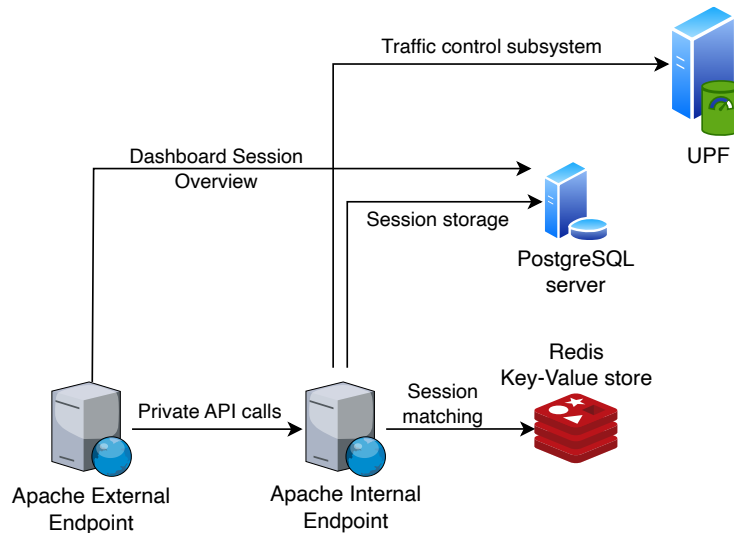


Figure 3.4: A first architectural overview outlining the presence of the TRAFFIC CONTROLLER

3.5. Deploying OpenAirInterface

As anticipated, OPENAIRINTERFACE (OAI) is an open-source initiative that provides 3GPP-compliant software implementations of both the 5G Core Network, Radio Access Network and UEs.

We will now deploy OAI's GNB HELM CHART, a containerized digital signal processing suite implementing the 5G NR base station responsible for the radio interface towards UEs and 5G Core Network.

3.5.1. Hardware paradigms: Ettus Research USRPs

ETTUS RESEARCH is a leading company that designs and manufactures **Software Defined Radio** (SDR) systems.

We will shortly make extensive use of ETTUS RESEARCH's Universal Software Radio Peripherals (USRPs):

- **USRP B210**: it is a compact and affordable bus-powered (USB) Software Defined Radio, it supports an extensive frequency range (70 MHz to 6 GHz), bandwidth (up to 40 MHz in our use-case) and 2×2 Multiple-Input, Multiple-Output (MIMO).
- **USRP N300**: it is a high-performance Software Defined Radio designed for the deployment of high-reliability advanced networks. The most notable upgrade for our solution is bandwidth, which can now reach 100 MHz and connectivity, which now employs high-speed networking via dual 10 GbE SFP+ ports.

Both of them will make use of supporting USRP Hardware Drivers (UHD), automatically retrieved at bootstrap time.

The role of Multus

As mentioned, MULTUS is a meta-CNI plugin for KUBERNETES allowing for **multiple networking interfaces to be attached to a single POD**.

OAI's HELM CHART defines a `NetworkAttachmentDefinition` custom resource which can be easily customized via templating variables in `values.yaml`.

```

1  ...
2  multus:
3    enabled: true
4    interfaces:
5      - name: "ru"
6        hostInterface: "enp65s0f0np0"
7        ipAdd: "192.168.50.100"
8        netmask: "24"
9        enabled: true
10       mtu: 9000
11       type: macvlan
12       mode: "bridge"
13     - name: "ru2"
14       hostInterface: "enp65s0f1np1"
15       ipAdd: "192.168.60.100"

```

```

16     netmask: "24"
17     enabled: true
18     mtu: 9000
19     type: macvlan
20     mode: "bridge"
21     ...

```

The main focus here is on host interfaces and the Maximum Transmission Unit (MTU), which is set to 9000 in spite of the traditional 1500 for typical Ethernet-based networks, to allow for better fragmentation and reduce overhead.

Hardware passthrough and security context

Radio access workloads are inherently hostile to standard container isolation mechanisms: they require **direct hardware access** and real-time scheduling routines.

The B210 USRP is a USB-based Software Defined Radio, which should therefore be mapped to the correct group of device files on the containerized environment [35]:

```

1     ...
2     {{- if eq .Values.config.radio "b2xx" }}
3         - name: radio
4           mountPath: /dev/bus/usb/
5         ...
6     {{- if eq .Values.config.radio "b2xx" }}
7         - name: radio
8           hostPath:
9             path: /dev/bus/usb/

```

Both deployments on B210 and N300 USRPs share a permissive security context:

- **privileged capabilities**: required for hardware access, memory mapping and bypassing standard isolation profiles.
- **SYS_NICE**: crucial for the setup of real-time scheduling classes/priorities via the Kernel.
- **run as root**: instantiating specific raw sockets and memory buffers requires root access.

Dynamic configuration patching

A significant challenge of configuring tens of network functions is the management of IP endpoints.

Thanks to Multicast DNS and the standardization of service names in KUBERNETES, they can be automatically resolved and patched into the HELM template configuration.

```

1 yq -i '
2   .Active_gNBs[0] = strenv(GNB_NAME) |
3   .gNBs[0].gNB_name = strenv(GNB_NAME) |
4   .gNBs[0].tracking_area_code = strenv(TAC) |
5   .gNBs[0].plmn_list = strenv(PLMN_LIST) |
6   .gNBs[].plmn_list |= (from_yaml) |
7   {{- if .Values.config.enableE2 }}
8   .e2_agent.near_ric_ip_addr = strenv(RIC_IP) |
9   {{- else }}
10  del(.e2_agent) |
11  {{- end }}
12  .gNBs[0].amf_ip_address[0].ipv4 = strenv(AMF_IP) |
13  .gNBs[0].NETWORK_INTERFACES.GNB_IPV4_ADDRESS_FOR_NG_AMF = strenv(N2_IP) |
14  .gNBs[0].NETWORK_INTERFACES.GNB_IPV4_ADDRESS_FOR_NGU = strenv(N3_IP)
15  ' /tmp/config_patch/gnb.yaml

```

In which we also highlight the presence of the E2 interface, allowing telemetry streaming and policy control through XAPPS deployed on the Near-Real-Time RIC (such as FLEXRIC, which we will shortly analyze in detail).

3.5.2. Observability through Vector

VECTOR is a high-performance, open-source **observability data pipeline for containerized environments**.

It is written entirely in RUST, providing memory safety guarantees without garbage collection, resulting in a more predictable behaviour and footprint, making it the perfect fit for our solution.

Architecture and configuration

VECTOR can be deployed as an aggregator (centralized cluster service to receive data from downstream agents) or as an agent, operating directly on compute nodes generating

the data.

In our configuration, it runs directly on nodes where the GNB PODS are scheduled. It mounts the **node's container logs and processes them** directly.

This results in the following snippet:

```

1  role: Agent
2  customConfig:
3      ...
4  sources:
5      gnb_logs:
6          type: "kubernetes_logs"
7          # The name label is our discriminant here
8          extra_label_selector: "app.kubernetes.io/name in (oai-gnb,
                                ↪ oai-gnb2, oai-gnb3)"

```

The `kubernetes_logs` source is a highly specialized component: it interfaces directly with the KUBERNETES API via service accounts to discover PODS, map container IDs and enrich incoming lines with related metadata (namespace, labels, annotations, etc). VECTOR will allocate file watchers to PODS labelled with the specified selectors.

Vector Remap Language

The core logic resides in the `process_telemetry` transformation, which uses re-mapping. This executes a script written in VECTOR REMAP LANGUAGE (VRL), an **expression-oriented language specifically compiled for observability data transformation** [28].

It is type-safe and provides a set of guarantees through the creation of an abstract syntax tree.

We used a series of conditional blocks based on specific text heuristics to determine the log line type. If a match is found, parsing is performed and a resulting map will contain extracted values.

Identity Mapping (UE-ID to RNTI)

```

1  ...
2  if starts_with(string!(app_name), "oai-gnb") {
3      # Association between UE-ID and C-RNTI (see specific mapper)

```

```

4     parsed_map = parse_regex(clean_msg, r'UE ID (?P<ue_id>\d+) RNTI
    ↪ (?P<rnti>[0-9a-fA-F]+)') ?? {}
5     if !exists(parsed_map.rnti) {
6         parsed_map = parse_regex(clean_msg, r'UE RNTI (?P<rnti>[0-9a-fA-F]+)
    ↪ CU-UE-ID (?P<ue_id>\d+)') ?? {}
7     }
8     if exists(parsed_map.rnti) && exists(parsed_map.ue_id) {
9         .metric_type = "gnb_ue_id_mapping"
10        .rnti = parsed_map.rnti
11        .ue_id = parsed_map.ue_id
12        matched = true
13    }
14    ...

```

In 5G, the Cell Radio Network Temporary Identifier (C-RNTI) is a 16-bit value assigned dynamically to a UE by the gNB.

Its role is very similar to MAC addresses in Ethernet-based networks, while the UE-ID is instead a logical sequential identifier to the Core Network for the current session.

RRC State Tracking

```

1     ...
2     if contains(clean_msg, "Received RRCSetupComplete") {
3         parsed = parse_regex(clean_msg,
    ↪ r'RNTI\s+(?P<rnti>[0-9a-fA-F]+)\s+Received RRCSetupComplete') ?? {}
4         if exists(parsed.rnti) {
5             .metric_type = "gnb_rrc_connected"
6             .rnti = parsed.rnti
7             matched = true
8         }
9     }
10
11    # Captures disconnections
12    if contains(clean_msg, "Remove NR rnti") {
13        parsed = parse_regex(clean_msg, r'Remove NR rnti 0x(?P<rnti>[0-9a-fA-F]+)')
    ↪ ?? {}
14        if exists(parsed.rnti) {
15            .metric_type = "gnb_rrc_disconnected"
16            .rnti = parsed.rnti
17            matched = true

```

```

18     }
19 }
20 ...

```

RRCSetupComplete and Remove NR rnti messages help the downstream backend to construct a state machine tracking exactly when a device enters in an RRC_CONNECTED state and when it drops back to idle or disconnects.

RF Signal Quality (RSRP)

```

1  if contains(clean_msg, "average RSRP") {
2      parsed = parse_regex(clean_msg, r'UE RNTI (?P<rnti>[0-9a-fA-F]+).*average
    ↪ RSRP\s+(?P<rsrp>[-0-9]+)' ?? {}
3      if exists(parsed.rsrp) {
4          .metric_type = "gnb_signal_quality"
5          .rnti = parsed.rnti
6          .rsrp_dbm = to_int!(parsed.rsrp)
7          matched = true
8      }
9  }

```

The Reference Signal Received Power (RSRP) is the most critical metric for RF coverage. We capture the received integer value and ensure proper serialization.

Channel state, modulation and throughput

```

1  ...
2  # MCS and SNR in UL (the same is performed for MCS-DL)
3  if contains(clean_msg, "ulsch_rounds") {
4      parsed = parse_regex(clean_msg, r'UE (?P<rnti>[0-9a-fA-F]+):
    ↪ ulsch_rounds.*MCS
    ↪ \(\d+\)\s+(?P<mcs_ul>\d+)\s+\(Qm.*SNR\s+(?P<snr>[-0-9\.]+)\s+dB' ?? {}
5      if exists(parsed.mcs_ul) {
6          .metric_type = "gnb_mcs_ul"
7          .rnti = parsed.rnti
8          .mcs_ul = to_int!(parsed.mcs_ul)
9          if exists(parsed.snr) {
10             .snr_db = to_float!(parsed.snr)
11         }
12         matched = true

```

```

13     }
14 }
15 if contains(clean_msg, "MAC:    TX") {
16     parsed = parse_regex(clean_msg, r'UE
    ↪  (?P<rnti>[0-9a-fA-F]+):\s+MAC:\s+TX\s+(?P<tx_bytes>\d+)\s
17         +RX\s+(?P<rx_bytes>\d+)') ?? {}
18     if exists(parsed.tx_bytes) {
19         .metric_type = "gnb_mac_throughput"
20         .rnti = parsed.rnti
21         .tx_bytes = to_int!(parsed.tx_bytes)
22         .rx_bytes = to_int!(parsed.rx_bytes)
23         matched = true
24     }
25 }
26 ...

```

We finally extract Modulation and Coding Schemes (MCS) for both Downlink (DL) and Uplink (UL), the Signal-to-Noise Ratio (SNR) and MAC layer throughput (TX and RX bytes per RNTI), which allows us to skip deep packet inspection on the UPF at this stage.

3.5.3. The sink: HTTP Webhook

Details extracted through VECTOR are **encoded in JSON** and **forwarded to an HTTP Webhook**, resolved through standard KUBERNETES DNS and running on the cluster as a containerized APACHE WEB SERVER installation.

```

1  ...
2  # Sinks (JSON output receivers)
3  sinks:
4      php_webhook:
5          type: "http"
6          inputs:
7              - "process_telemetry"
8          uri: "http://apache-internal-svc.apache-internal.svc.cluster.local
9              /api/webhook.php"
10         method: "post"
11         encoding:
12             codec: "json"
13         batch:

```

```

14         max_events: 5000
15         timeout_secs: 5

```

APACHE WEB SERVER is an open-source HTTP Server developed by the APACHE SOFTWARE FOUNDATION.

It operates as a highly-configurable and modular daemon supporting dynamic web pages extensions like PHP or JAVA TOMCAT.

PHP: HYPERTEXT PREPROCESSOR (PHP), is an interpreted scripting language for dynamic web pages and server-side web applications (e.g. Content Management Systems like WordPress).

Being an interpreted language, its deployment allows for a high degree of flexibility and malleability, which is ideal for our use-case.

Our webhook is deployed as a **stateless containerized service within the KUBERNETES cluster, exposed internally via an HTTP Service**. It operates at the intersection of three major domains:

- VECTOR: the logging plane, an upstream agent providing batches of events periodically.
- REDIS: used for high-speed identity mapping lookups, active session caching and serving transient radio metrics.
- POSTGRESQL: the persistent storage layer, used for immutable audit trails of UE sessions and durable storage of traffic volume metrics.

REMOTE DICTIONARY SERVER (REDIS) is a NOSQL open-source key-value data store created in Italy, mainly used as a high-performance caching system or message broker.

POSTGRESQL is an open-source Object-Relational Database Management System (ORDBMS) widely adopted in the industry for its exceptional reliability and flexibility.

The identity resolution problem: a first introduction

A fundamental challenge to our telecommunications observability framework is the **fragmentation of identifiers** across the stack:

- **Layer 2 (MAC):** GNB logs metrics (SNR, MCS, throughput) against the Cell Radio Network Temporary Identifier (C-RNTI), a 16-bit hexadecimal value that changes every time the UE connects or transition states.
- **Layer 3 and up (Core):** billing and subscriber management rely instead on the

International Mobile Subscriber Identity (IMSI).

Our webhook relies on a **custom mapping solution** (which we will describe in detail in the following sections), that populates REDIS with the correlation between an active RNTI and its corresponding IMSI.

A single customer (UE) can therefore be precisely identified across both **terminated and future sessions**.

When processing VECTOR log entries, if an RNTI is encountered for the first time, REDIS is queried for the IMSI mapping and POSTGRES is then informed of aggregate traffic cumulatives.

We manage both the first insertion, updates and final confirmation of records for customer sessions: this is, essentially, a state machine driven by VECTOR log events operating through SQL queries and REDIS **set/get** operations.

```

1  ...
2  // a session is inserted ...
3  $qInsertSess = $pdo->prepare(
4
5      ↪  'INSERT INTO polimi.CustomerSession (imsi, ue_id, amf_id, rnti, ipv4, "start")
6      VALUES (:imsi, :ue_id, :amf_id, :rnti, :ipv4, :start) RETURNING id',
7  );
8  // updated ...
9  $qUpsertPerf = $pdo->prepare('
10     INSERT INTO polimi.CustomerPerformance (session_id, imsi, tx_kbytes, rx_kbytes)
11     VALUES (:session_id, :imsi, :tx, :rx)
12     ON CONFLICT (session_id, imsi) DO UPDATE
13     SET tx_kbytes = EXCLUDED.tx_kbytes, rx_kbytes = EXCLUDED.rx_kbytes
14 ');
15 // ... and finalized
16 $qCloseSess = $pdo->prepare(
17     'UPDATE polimi.CustomerSession SET "end" = :end_time WHERE id = :db_id
18     AND "end" IS NULL',
19 );
20 ...

```

Handling cumulative metrics and real-time feeds

To lift excessive and continuous processing on REDIS and POSTGRES sides, we manage throughput and radio metrics aggregation in memory, before committing them to the respective long-term or short-term storage solution.

```

1 // Performance metrics cumulative
2 if ($db_id && $type === "gnb_mac_throughput") {
3     $performance_updates[$db_id] = [
4         "imsi" => $imsi,
5         "tx" => (int)floor($e["tx_bytes"] / 1024),
6         "rx" => (int)floor($e["rx_bytes"] / 1024),
7     ];
8 }
9
10 // Dashboard metrics update
11 if ($imsi) {
12     $metrics_key = "dashboard:metrics:{$imsi}";
13     $live_data = [];
14
15     if ($type === "gnb_signal_quality") {
16         $live_data["rsrp"] = $e["rsrp_dbm"];
17     }
18     if ($type === "gnb_mcs_dl") {
19         $live_data["mcs_dl"] = $e["mcs_dl"];
20     }
21     if ($type === "gnb_mcs_ul") {
22         $live_data["mcs_ul"] = $e["mcs_ul"];
23         if (isset($e["snr_db"])) {
24             $live_data["snr"] = $e["snr_db"];
25         }
26     }
27     if ($type === "gnb_mac_throughput") {
28         $live_data["tx_kb"] = (int)floor($e["tx_bytes"] / 1024);
29         $live_data["rx_kb"] = (int)floor($e["rx_bytes"] / 1024);
30     }
31
32     if (!empty($live_data)) {
33         $redis->hMSet($metrics_key, $live_data);
34         $redis->expire($metrics_key, 600);

```

```

35     }
36 }

```

We finally highlight the usage of an explicit 10-minutes Time-To-Live (TTL) for REDIS metrics to automatically discard useless real-time metrics collected for currently disconnected or inactive users.

3.5.4. Deploying FlexRIC

The O-RAN architecture decouples the Control Plane and the underlying COTS hardware, allowing for Software-Defined abstractions inspired by recent trends in the telecommunications industry. At the heart of such ecosystem is the NEAR-RT RIC, a **software-defined platform allowing for the execution of third party applications (xApps)**, radio resource optimization and traffic inspection with **small latencies**.

The E2 interface is the standard communication channel between the RIC and E2 nodes (GNB and related components) on which the RIC receives **telemetry details (KPM)** and **exchange control messages (RC)**.

OPENAIRINTERFACE FLEXRIC is an open-source and highly modular Near-RT RIC developed by the OPENAIRINTERFACE ALLIANCE. His main strengths are its flexibility and efficiency, making it an ideal fit for our solution [33].

Since we need to provide our customized environment, we developed a **custom containerized environment** based on the most common one being developed and maintained as part of the project.

Building a custom FlexRIC environment

```

1  ARG BASE_IMAGE=ubuntu:jammy
2
3  FROM $BASE_IMAGE AS oai-flexric
4  ENV DEBIAN_FRONTEND=noninteractive
5  ENV TZ=Europe/Rome
6  # ... (omitted dependencies provisioning)
7  RUN git clone "https://gitlab.eurecom.fr/mosaic5g/flexric.git"
8  WORKDIR /flexric
9
10 # To ensure reproducibility, a fixed commit had to be chosen:
    ↪ 7805d63c3157dee630f11d6eef2f5ed540f5209a have been made on 05/05/2026

```

```
11 RUN git reset --hard 7805d63c3157dee630f11d6eef2f5ed540f5209a
12
13 # Inject our custom xApp in the build cycle!
14 RUN mkdir -p /flexric/examples/xApp/c/kpm_custom/
15 COPY CMakeLists.txt /flexric/examples/xApp/c/kpm_custom/CMakeLists.txt
16 COPY kpm_custom.c /flexric/examples/xApp/c/kpm_custom/kpm_custom.c
17
18 RUN echo "add_subdirectory(kpm_custom)" >>
    ↪ /flexric/examples/xApp/c/CMakeLists.txt
19
20 # ... (compile and link libraries)
21 COPY entrypoint.sh /usr/local/flexric/entrypoint.sh
22 RUN chmod +x /usr/local/flexric/entrypoint.sh
```

In which we highlight the use of a fixed FLEXRIC stable version and the **injection of our custom xAPP in the build process**.

A custom **entrypoint** is also being used: this is the target that will be run upon creation of the final containerized environment.

Here, it will just run FLEXRIC and our xAPP in parallel.

An observability xApp in KPM

To leverage the observability strengths of KPM xAPPS, we developed a comprehensive telemetry aggregator for radio and performance metrics.

Our solution decodes encoded payloads coming from 5G base stations (OAI GNB), aggregates metrics over a sliding window and flushes summarized data to an internal HTTP webhook.

We would therefore like to set up three primary responsibilities:

- **Subscriptions management:** discovering connected E2 nodes and establishing subscription contracts for specific KPM reports.
- **Data extraction and aggregation:** parsing the incoming E2 Indication messages, matching specific metric names and accumulating values in a stateful array.
- **Data exfiltration:** periodically converting the aggregated state into a JSON payload and transmitting it over HTTP to the same APACHE WEB SERVER instance we used before.

We then begin by iterating over connected E2 nodes and check whether they support

KPM (id = 2).

If supported, the XAPP must negotiate how the data should be sent: this is done by generating a **subscription payload** defining an event trigger (aggregate and send data periodically) and an action definition (the specific metrics to report):

```

1  int const KPM_ran_function = 2; // KPM Service Model
2
3  for (size_t i = 0; i < nodes.len; ++i) {
4      e2_node_connected_xapp_t *n = &nodes.n[i];
5      size_t const idx = find_sm_idx(n->rf, n->len_rf, eq_sm, KPM_ran_function);
6      ...

```

We then parse and analyze the message, the UE list and therefore retrieve **measurement data and related records**.

```

1  ...
2  char *name_str = cp_ba_to_str(info_item.meas_type.name);
3  char *name_unit = get_meas_unit(name_str);
4
5  ...
6
7  else if (strcmp(name_str, "RRU.PrbTotDl") == 0)
8      ue_stats[idx].PrbTotDl += val;
9  else if (strcmp(name_str, "RRU.PrbTotUl") == 0)
10     ue_stats[idx].PrbTotUl += val;
11  ...

```

A raw POSIX Socket client is finally used for raw HTTP communication and payload construction.

Upon receipt of the forwarded payload, the PHP ingestion API deployed on the APACHE WEB SERVER will then:

- Set up a connection to the cluster's REDIS endpoint for short-term storage.
- Parse the aggregated data coming from the XAPP.
- Set up a REDIS entry using, as an identifier the AMF-UE-NGAP-ID, a sequential and unique identifier assigned by the AMF after completion of the Authentication procedure.

This uniquely identifies a session inside the whole deployment in which the same

Core Network is being used: it is completely **independent of the tenant in which the session is registered**.

- Iterate over all UEs creating a structured HASH entry in REDIS.

```

1  ...
2  foreach ($data["ue_list"] as $ue) {
3      $amf_id = $ue["amf_ue_ngap_id"];
4      $redis_key = "dashboard:kpm:{$amf_id}";
5
6      $hash_data = [
7          "timestamp" => $timestamp,
8          "ran_ue_id" => $ue["ran_ue_id"],
9          "PdcpsduVolumeDL_Mb" => $ue["PdcpsduVolumeDL_Mb"],
10         "PdcpsduVolumeUL_Mb" => $ue["PdcpsduVolumeUL_Mb"],
11         "RlcSduDelayDl_s" => $ue["RlcSduDelayDl_s"],
12         "UETHpDl_kbps" => $ue["UETHpDl_kbps"],
13         "UETHpUl_kbps" => $ue["UETHpUl_kbps"],
14         "PrbTotDl_pct" => $ue["PrbTotDl_pct"],
15         "PrbTotUl_pct" => $ue["PrbTotUl_pct"],
16         "PacketLossRateDl_pct" => $ue["PacketLossRateDl_pct"],
17         "PdcpsDropRateDl_pct" => $ue["PdcpsDropRateDl_pct"],
18     ];
19
20     $redis->hMSet($redis_key, $hash_data);
21     // Zombie sessions are discarded within a 25 second timeout
22     $redis->expire($redis_key, 25);
23     ...

```

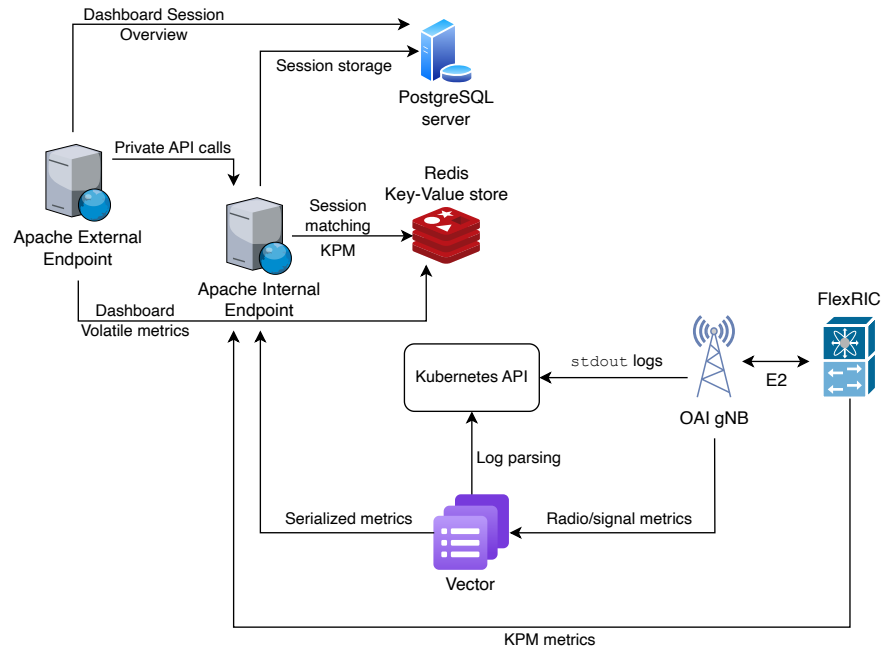


Figure 3.5: The resulting architecture, where the use of VECTOR and FLEXRIC is highlighted

3.6. The identity resolution problem

As anticipated, **cross-layer UE identifiers correlation** is a fundamental challenge in 5G networks: highly optimized protocols and disaggregation of Network Functions have produced an innovative and competitive environment, while creating some disadvantages from the architectural point of view.

Let's sum up the main players in this space:

- **gNB**: the 5G base station, managing the radio link.
- **AMF**: Access and Management Function, the entry point to the Control Plane of a 5G network.
It manages authentication, registration and mobility.
- **SMF**: Session Management Function, handles data sessions and IP address allocation.

Table 3.1: The identifiers in a 5G deployment

Identifier	Meaning	Visibility/Context
RNTI	A hexadecimal string (e.g. 0x12ab) assigned by the GNB for radio link scheduling.	Only known to the GNB and UE. The 5G Core has no conception of RNTIs.
IMSI	The permanent Subscription ID (stored on the physical SIM card)	Known to the AMF and SMF. The GNB has no conception of this ID for disaggregation security.
NGAP IDs	RAN_UE_NGAP_ID and AMF_UE_NGAP_ID. Used to identify a logical connection between the GNB and AMF.	Known to both GNB and AMF, acting as the crucial bridge to link RAN and Core identifiers.
IPv4 addresses	The IP address assigned to the UE for internet access.	Known to the SMF and UPF, linked to NGAP IDs.

We aim at resolving this challenge by exploiting the **privileged access we have to the infrastructure as a whole**, by analyzing and dissecting the standard output (stdout) and log entries provided by the AMF, SMF and GNB via the KUBERNETES API.

After sanitization, we compile **a set of Regular Expressions (regex) objects to parse log lines**, in the same way we did for VECTOR when deploying OAI's GNB.

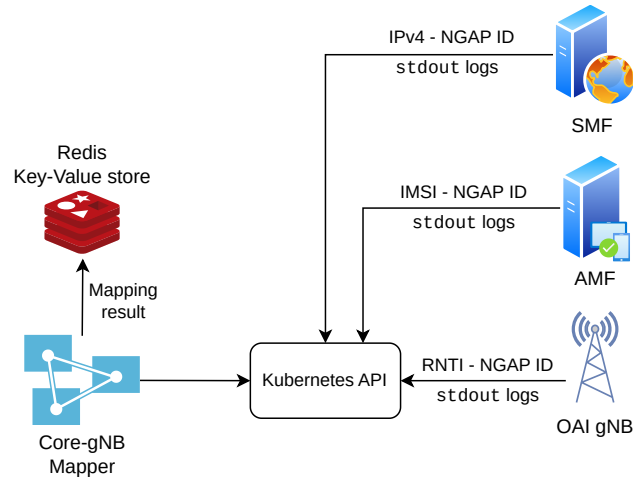


Figure 3.6: A partial infrastructural overview showing the GNB-CORE mapper in action

3.6.1. Advanced connection management and infrastructure monitoring

Re-establishments

In a 5G network, whenever a UE momentarily loses signal due to an obstacle, the radio link drops. The Core Network still keeps track of the session and, if the UE quickly re-connects to the GNB, it will grant the UE **a completely new RNTI because of the freshly-created session** on the GNB.

Without a special logic managing RRC Re-establishments, our mapper **would lose correlation** since the old RNTI still maps as an active session to the IMSI, while the newly-created RNTI appears as unknown and discarded.

```

1 match_reest = regex_reestablishment.search(clean_line)
2 if match_reest:
3     new_rnti = match_reest.group("new_rnti").lower()
4     old_rnti = match_reest.group("old_rnti").lower()
5     # ... reverse-lookup in local dictionary ...
6     if found_key:
7         match_cache[found_key]["rnti"] = new_rnti
8         delete_from_redis(f"{gnb_tenant}:{old_rnti}:mapping")
9         logging.info(
10             f"[{found_key}] RRC Reestablishment:
  
```

```
11 RNTI updated ({old_rnti} -> {new_rnti})")
```

The same pattern is, of course, valid for all traditional matching routines.

Infrastructure monitoring: Kubernetes event watcher

Moving from the parsing logic to the infrastructure orchestration, we would like to create a comprehensive event watcher which adapts to the varying environment: restarted, deleted or newly-running Pods, including both GNB and the 5G Core Network.

```
1 v1 = client.CoreV1Api()
2 while True:
3     try:
4         w = watch.Watch()
5         for event in w.stream(v1.list_pod_for_all_namespaces):
6             pod = event["object"]
7             pod_name = pod.metadata.name
8             namespace = pod.metadata.namespace
9             app_name = ...
10
11             if event["type"] == "DELETED" or pod.status.phase != "Running":
12                 if pod_name in watched_pods:
13                     watched_pods.remove(pod_name)
14                     if "amf" in app_name:
15                         cleanup_all() # deleting the whole deployment session
16                     elif app_name in TENANT_NAMING.keys():
17                         cleanup_tenant(tenant_id) # deleting all zombie
18                             ↪ sessions...
19
20             continue
```

We must remember here the main assumption throughout our work: even if an indefinite number of tenants is operating on our infrastructure (cluster), **a single Core Network is being used across all of them.**

If this assumption drops, proper adjustments to the management of dangling sessions should be considered.

Kubernetes API access

To safely deploy a POD interfacing the KUBERNETES API, strict Role-Based Access Control (RBAC) must be configured:

```

1  apiVersion: v1
2  kind: ServiceAccount
3  metadata:
4      name: log-matcher-sa
5  ---
6  apiVersion: rbac.authorization.k8s.io/v1
7  kind: ClusterRole
8  metadata:
9      name: log-matcher-role
10 rules:
11     - apiGroups: ["" ]
12       resources: ["pods", "pods/log"]
13       verbs: ["get", "list", "watch"]
14 ---
15 ...

```

Of course, we employ the principle of least privileging: we allow only relevant actions (get, list, watch) on pods and their logs.

Aggregation and notes on sequentiality

We then finally **aggregate all relevant details** and publish them through REDIS:

```

1  def check_and_publish(cache_key):
2      with lock:
3          data = match_cache.get(cache_key)
4          if not data:
5              return
6
7          if all(k in data for k in ["rnti", "imsi", "amf_id", "tenant"]):
8              rnti, tenant, ran_id = data["rnti"], data["tenant"],
8              ↪ cache_key.split("_")[1]
9              imsi = data["imsi"]
10             redis_key = f"{tenant}:{rnti}:mapping"
11             payload = {
12                 "amf_id": str(data["amf_id"]),
13                 "ran_ue_id": str(ran_id),
14                 "IMSI": imsi,
15                 "RNTI": rnti,

```

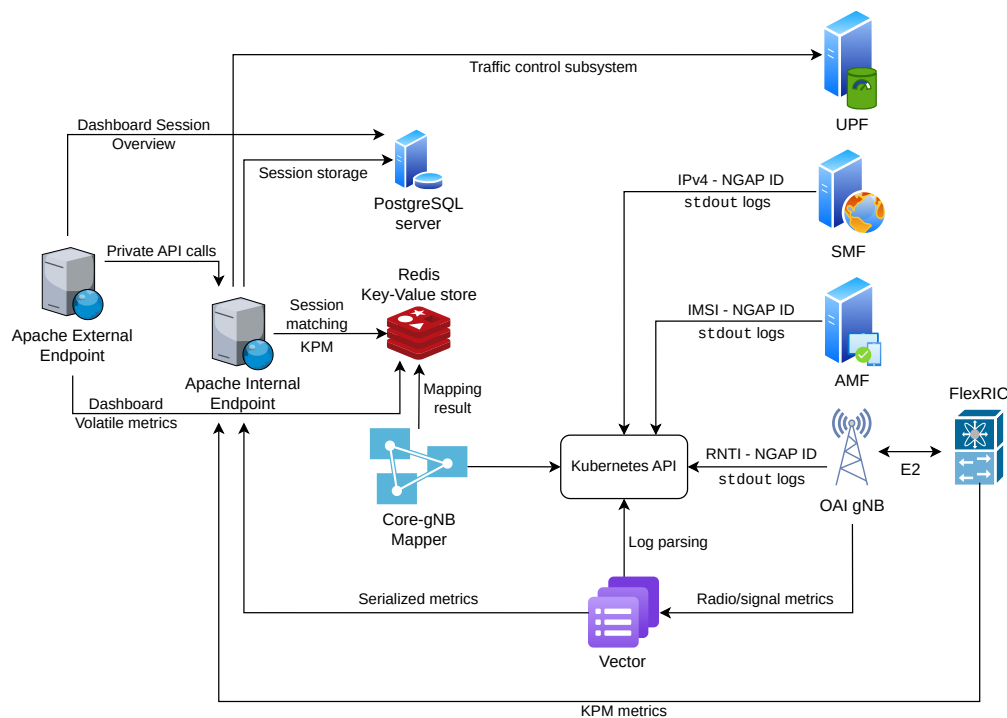
20

as a guarantee of completeness.

3.7. A live dashboard for end-users

solutions.

and traffic control.



and integrated in our solution

3.7.1. PostgreSQL database

In the following, we will detail the **Entity-Relationship (E-R) model diagram** and **SQL schema** for long-term data storage in the Neutral Host management platform.

E-R diagram

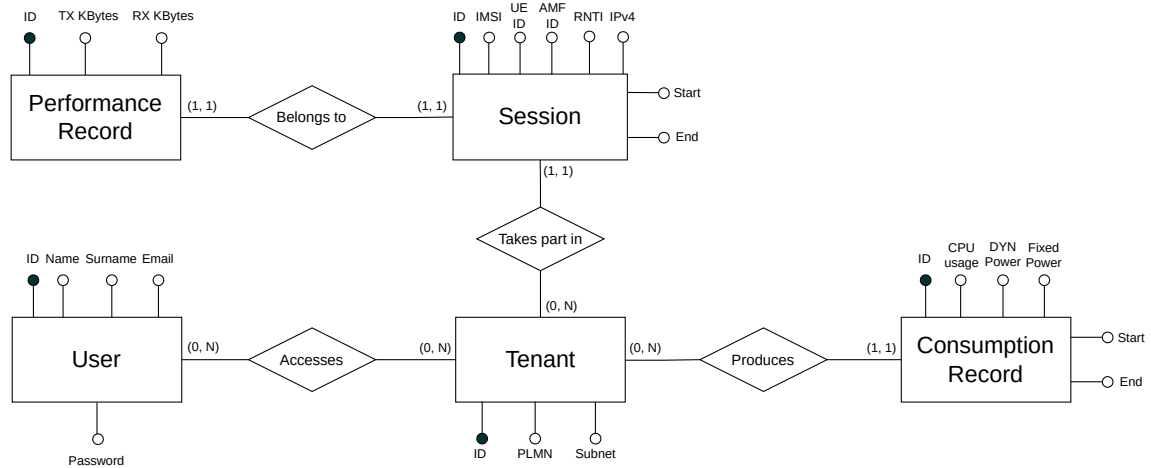


Figure 3.8: E-R model diagram for the Neutral Host management platform

SQL Schema

```

1 CREATE TABLE CustomerSession (
2     id SERIAL NOT NULL,
3     imsi VARCHAR(15) NOT NULL,
4     ue_id INT NOT NULL,
5     amf_id INT NOT NULL,
6     rnti VARCHAR(20) NOT NULL,
7     ipv4 VARCHAR(15),
8     "start" TIMESTAMP NOT NULL,
9     "end" TIMESTAMP,
10    PRIMARY KEY (id, imsi)
11 );
12
13 CREATE TABLE
14     CustomerPerformance (
15         imsi VARCHAR(15) NOT NULL,
16         session_id INT NOT NULL,
  
```

```

17         tx_kbytes INT NOT NULL,
18         rx_kbytes INT NOT NULL,
19         PRIMARY KEY (imsi, session_id),
20         FOREIGN KEY (imsi, session_id) REFERENCES CustomerSession (imsi, id) ON
           → UPDATE CASCADE ON DELETE CASCADE
21     );
22
23 CREATE TABLE "User"
24 (
25     id          VARCHAR(200) NOT NULL PRIMARY KEY,
26     name        VARCHAR(255) NOT NULL,
27     surname     VARCHAR(255) NOT NULL,
28     email       VARCHAR(255) NOT NULL,
29     salt        VARCHAR(60)  NOT NULL,
30     SHAPassword VARCHAR(64)  NOT NULL
31 );
32
33 CREATE EXTENSION IF NOT EXISTS pgcrypto;
34
35 CREATE OR REPLACE PROCEDURE createUser( ... )
36 LANGUAGE plpgsql
37 ...
38
39 CREATE TABLE Tenant
40 (
41     id          SERIAL PRIMARY KEY,
42     PLMN        VARCHAR(6) NOT NULL UNIQUE,
43     subnet      VARCHAR(18) NOT NULL UNIQUE
44 );
45
46 CREATE TABLE UserPermissions
47 (
48     id          SERIAL,
49     userId      VARCHAR(200) NOT NULL,
50     tenant      VARCHAR(6) NOT NULL,
51     PRIMARY KEY (id, userId, tenant),
52     FOREIGN KEY (userId) REFERENCES "User" (id)
53         ON UPDATE CASCADE ON DELETE CASCADE,
54     FOREIGN KEY (tenant) REFERENCES Tenant (PLMN)
55         ON UPDATE CASCADE ON DELETE CASCADE

```

```

56 );
57
58 CREATE TABLE TenantConsumption
59 (
60     id                SERIAL,
61     tenant            VARCHAR(6) NOT NULL,
62     cpu_usage         INT NOT NULL,
63     dynamic_watts     REAL NOT NULL,
64     fixed_watts       REAL NOT NULL,
65     start             TIMESTAMP NOT NULL,
66     "end"             TIMESTAMP NOT NULL,
67     PRIMARY KEY (id, tenant),
68     FOREIGN KEY (tenant) REFERENCES Tenant (PLMN)
69         ON UPDATE CASCADE ON DELETE CASCADE
70 );

```

With `createUser` being a stored procedure handling the cryptographic and hashing strategies behind the creation of user entries.

In this realm, the `User` of the Neutral Host management platform is the responsible entity for a customer of the Neutral Host, which therefore is enabled on controlling a pre-defined set of tenants.

3.7.2. Redis key-value store

In the following, a schematic view of key-value pairs present in REDIS for short-term storage purposes.

Power Metrics		UE and Radio Metrics		Identity Resolution
dashboard:status:{tenant_id}	dashboard:power:{tenant_id}	dashboard:kpm:{amf_id}	dashboard:metrics:{imsi}	{tenant_id}:{rnti}:mapping
plmn_id	cores_used	timestamp	rsrp	RNTI
deployment	power_watts	ran_ue_id	mcs_dl	IMSI
allocated_cores	timestamp	PdcpSduVolumeDL_Mb	mcs_ul	ran_ue_id
fronthaul_core		PdcpSduVolumeUL_Mb	snr	amf_id
current_governor		RlcSduDelayDL_s	tx_kb	IPv4
base_governor		UETHpDL_kbps	rx_kb	
hw_error		UETHpUL_kbps		
active_users		PrbTotDL_pct		
timestamp		PrbTotUL_pct		
		PacketLossRateDL_pct		
		PdcpDropRateDL_pct		

Figure 3.9: Key-value pairs stored in REDIS

3.7.3. End-user interface

All collected metrics and control capabilities are made available to end-users (customers for the Neutral Host) via a Graphical User Interface (GUI), realized through **web programming technologies**.

In the following, we will show the resulting web interface, highlighting the most relevant interest points.

Global overview

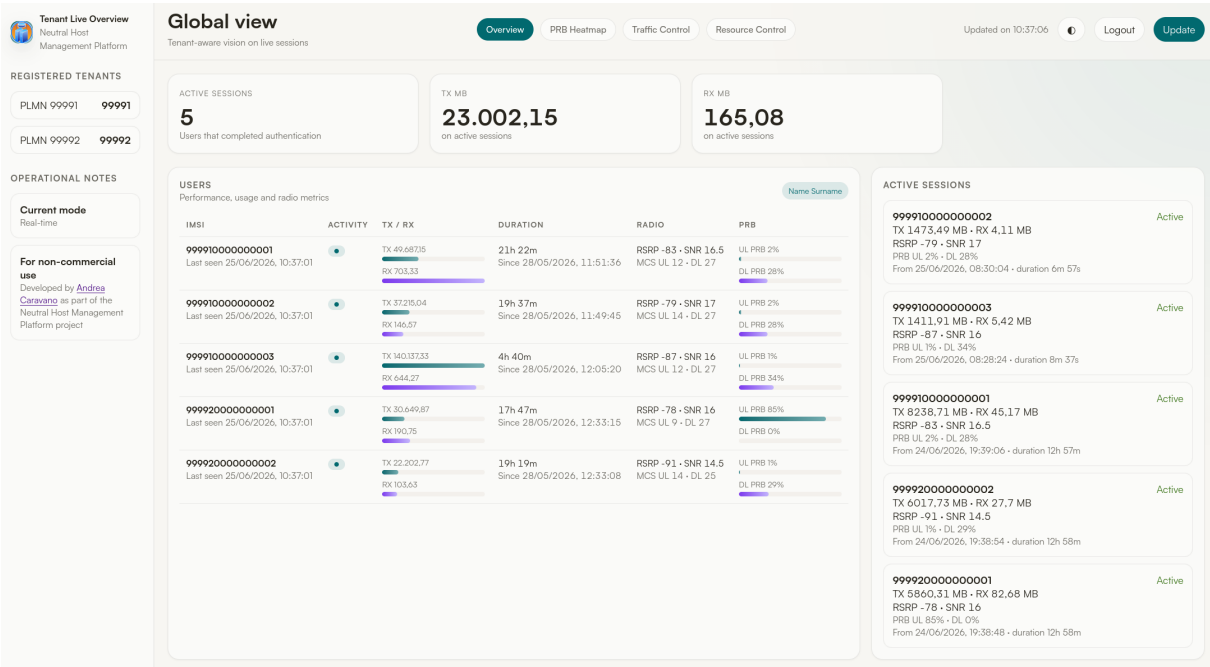


Figure 3.10: Tenant Live overview

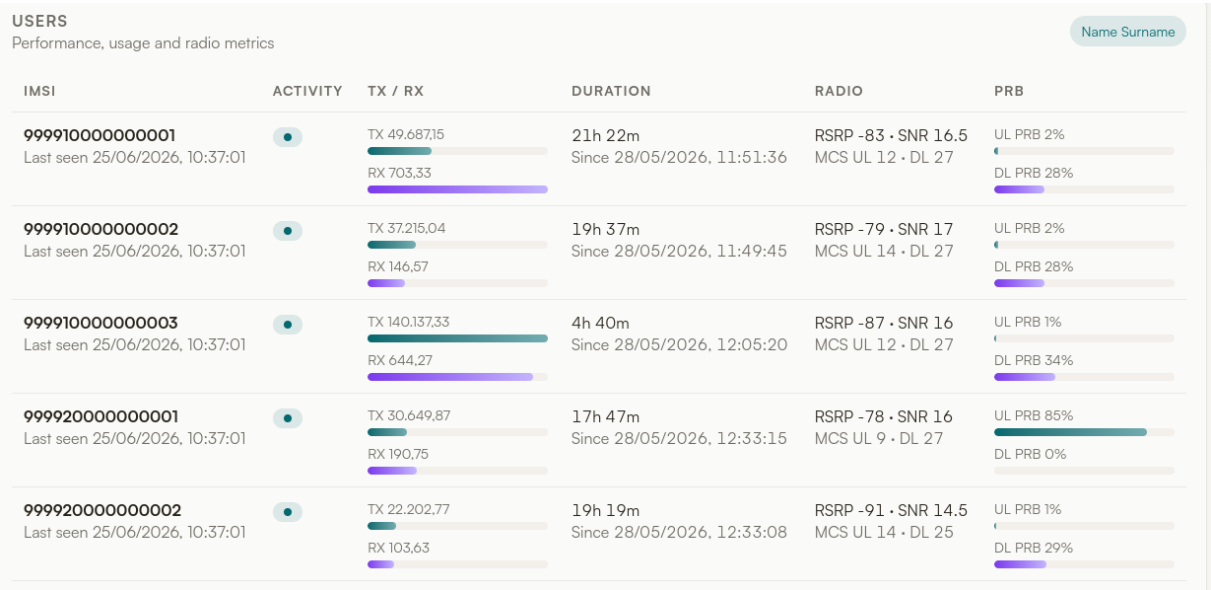


Figure 3.11: Historical performance and radio usage metrics on a per-UE basis

PRB usage heatmap

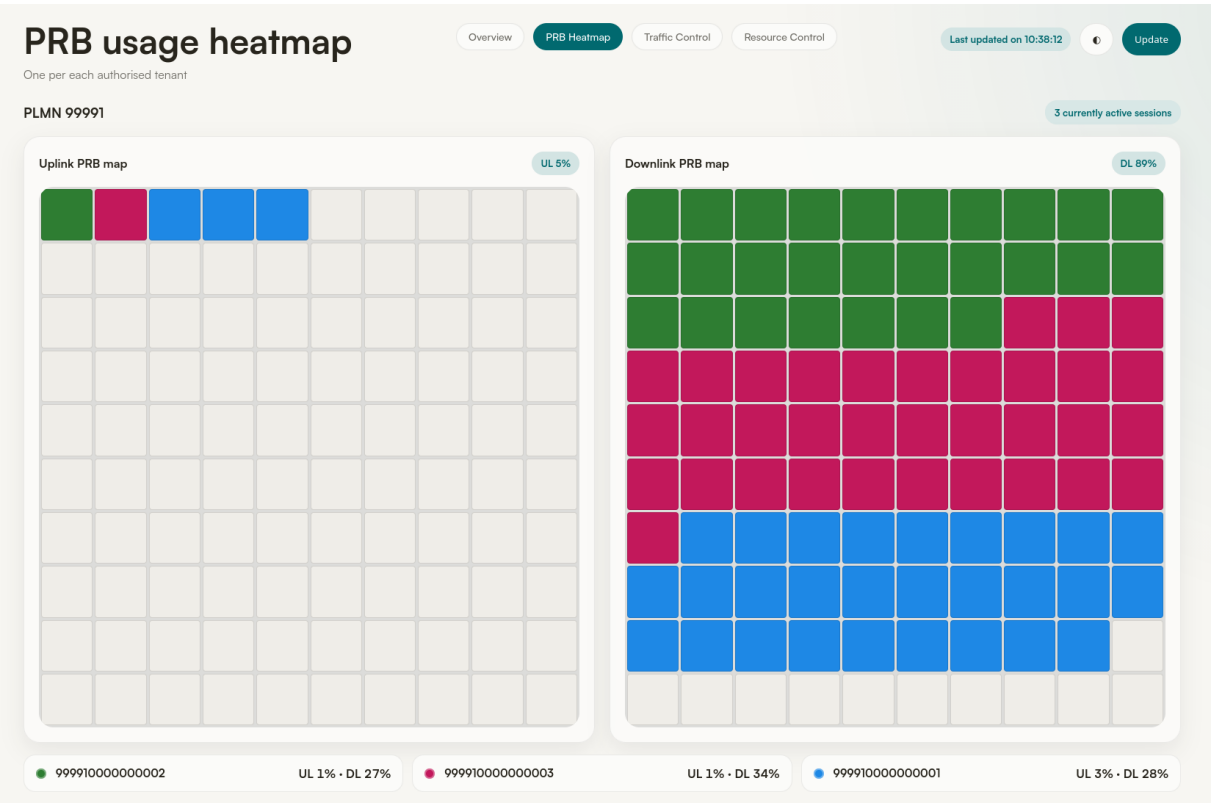


Figure 3.12: Live PRB usage heatmap for tenant n. 1



Figure 3.13: Live PRB usage heatmap for tenant n. 2

An inspection of the live PRB usage heatmap for both tenants allow us to clearly detect the current performance pattern: on tenant n. 1, a global Downlink (DL) throughput test is being executed, while on tenant n. 2 a single UE is performing a significant Uplink (UL) contribution.

Traffic control

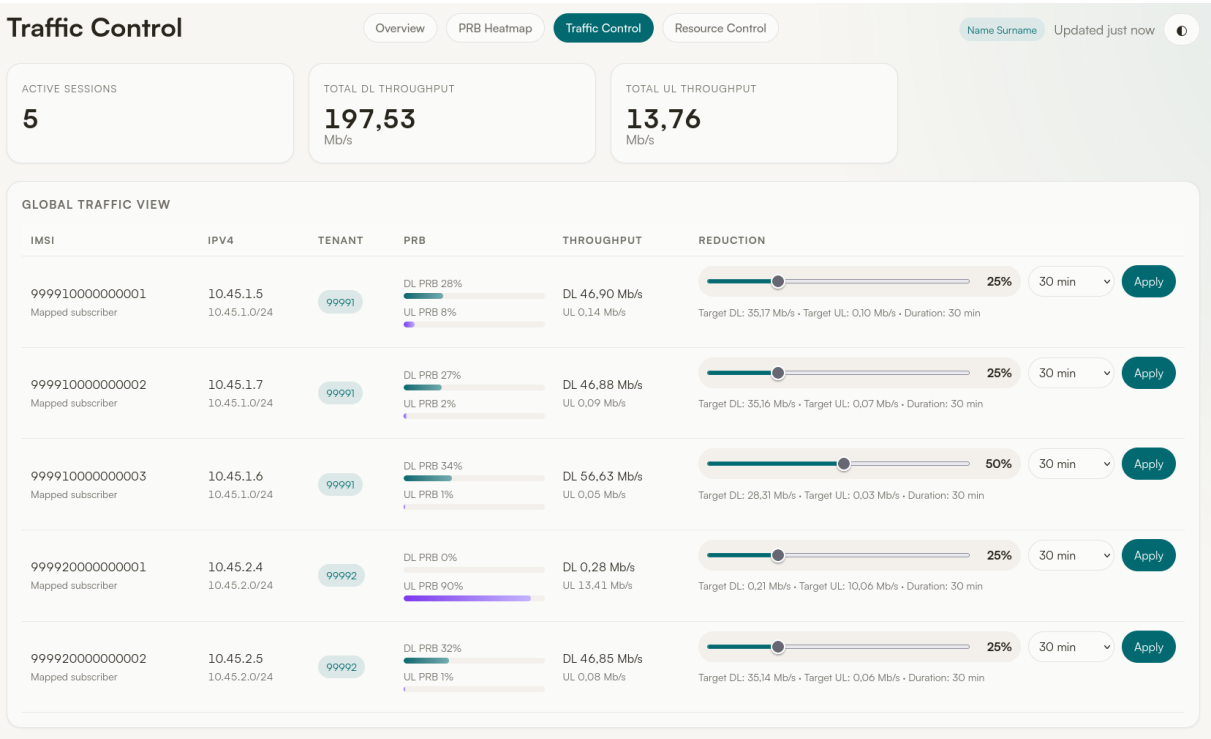


Figure 3.14: Live traffic overview

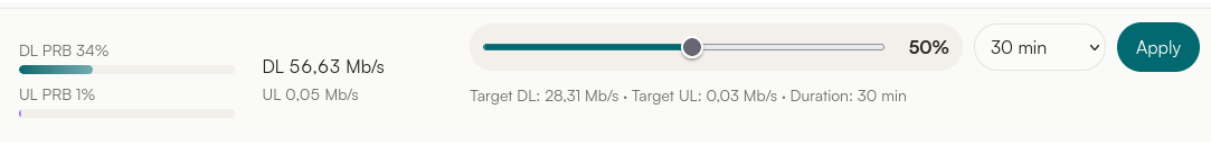


Figure 3.15: On each currently active session, we can enforce custom rate limiting strategies

4 | Experimental evaluation and use-cases

4.1. System profiling and energy tuning on AMD platforms

The transition to Cloud-Native RAN, enabled by O-RAN, shifts many elements of the processing to COTS hardware.

Architectures like INTEL XEON or AMD RYZEN are not inherently designed for continuous, deterministic radio processing.

In our 5G Neutral Host architecture, the system leverages some parametric optimization possible at both Kernel and run-time levels.

Our **laboratory testbed**, where we deployed our Single Node KUBERNETES cluster is as follows:

Laboratory testbed technical specifications

HP Z6 G5 A Workstation, SKU: 7D8C0AV

CPU: AMD Ryzen Threadripper PRO 7945WX x86_64, 12 cores-24 threads

RAM: 2 × 32 GB Micron Technology Registered DIMM 5600 MHz

2 × **Intel Ethernet Controller X710 for 10 GbE SFP+**

Realtek RTL8111/8168/8211/8411 PCIe Gigabit Ethernet Controller

NVIDIA T400 4GB PCIe

Samsung PM9A1 M.2 PCIe 4.0 NVME TLC Solid State Drive

4.1.1. Model-Specific Registers for Power Control

Historically, manipulation of CPU performance states and power limits in Linux was achieved by reading from and writing to **Model-Specific Registers** (MSR).

MSRs are specialized control registers within the x86 architecture. They are exposed to the Linux Kernel via the `/dev/cpu/*/msr` character device.

Writing specific hexadecimal payloads to the specific memory addresses (via `wrmsr`), we could theoretically **alter the Running Average Power Limit** (RAPL) parameters.

While RAPL via MSRs works reliably on most Intel platforms, it has proven unreliable for power enforcement on modern AMD Zen Architectures (Ryzen/EPYC platforms) [34].

Even if unofficial working solutions do exist [40], they are typically not yet updated to fully support or interpret correctly MSRs on our AMD Zen workstation.

Moreover, in typical commercial systems, protection of MSRs writes could also be present and inhibit coherent manipulation of parameters of interest [29].

4.1.2. AMD Host System Management Port

To address this issue, AMD provides the **Host System Management Port** (HSMP) in newer Zen architectures [20].

Unlike MSRs, which are arbitrary registers, AMD HSMP is a structured, mailbox-based **communication interface between the host Operating System** (Linux, in our case) and the **System Management Unit** (SMU).

The SMU is a dedicated microcontroller embedded directly onto the AMD silicon die. It governs **powers, thermals and frequencies**.

Invoking an HSMP utility translates in providing a corresponding SMU action, which will request and enforce it at the hardware level.

Enforcing a deterministic power limit is as easy as invoking the `e_smi_tool`, which interfaces with the HSMP driver.

For example [17],

```
sudo e_smi_tool --setpowerlimit 0 100000
```

Indicates the request for **Socket 0** (housing the whole CPU, the only socket in our system) for a target power limit of $100000\text{ mW} = 100\text{ W}$.

The SMU will then clamp the Package Power Tracking limit to 100 W , dynamically reducing the voltage/frequency curve across all cores to ensure that the aggregate power draw of the silicon stays below the imposed threshold.

4.1.3. Telemetry and profiling via AMD μ Prof

Governing power limits is only half of our needs. To correctly understand how our 5G workload distributes load across the system, we require hardware telemetry. Standard Linux tools like `s-tui` rely on Kernel abstractions, while **the real hardware solution is AMD μ Prof, the all-in-one profiling and telemetry suite for AMD systems.**

μ Prof is available as a pre-compiled binary for most GNU/Linux distributions. It interfaces directly with the hardware Performance Monitoring Counters (PMC) and the SMU [16].

A 1-hour **time-chart** can be requested via:

```
sudo AMDuProfCLI timechart --event power,frequency,temperature --duration 3600
```

It will provide a CSV trace in `/tmp/`, which can then be easily parsed for analysis or policy enforcement, as we will shortly do when providing energy governance on our Neutral Host Management Platform.

- **timechart**: Instructs μ Prof to sample data at regular intervals and output a chronological time series in CSV format.
- **-event power,frequency,temperature**: we instruct the SMU to dump data for all three critical domains.
- **-duration 3600**: profiles the system for one hour. A larger profiling window allows for broader temperature stability, as we will learn when analyzing consumption measurements.

Analyzing the profile report

The resulting output a Comma Separated Value (CSV) **dataset containing relevant data points**. The CSV is structurally divided into metadata and records.

The header defines the sampling frequency (typically 1000 *ms*) and maps the column IDs to hardware sensors.

Profiled counters are, in our case:

- **threadX-core-effective-frequency**: this is the critical frequency metric. Unlike the base clock reported by the OS, the effective frequency is the average frequency the core actually sustained during the profiling interval (1 second in most cases).

- `socket0-package-power`: the aggregate power draw in Watts of the entire CPU socket.
- `coreX-power`: the specific wattage drawn by an individual physical core (as anticipated, cores and threads respect symmetry assignment rules).

Data rows represent **chronological snapshots of profiled counters**.

A Python analytical framework

To transform telemetry into actionable decisions, we created a **PYTHON JUPYTER notebook**.

We will extensively make use of its fundamentals when finalizing our Neutral Host Management Platform. For the time being, let's dissect its core capabilities.

```

1 def load_and_process(filepath):
2     # Detect start line dynamically
3     start_line = 0
4     with open(filepath, 'r') as f:
5         for i, line in enumerate(f):
6             if "PROFILE RECORDS" in line:
7                 start_line = i + 1
8                 break
9
10    # Load CSV
11    df = pd.read_csv(filepath, skiprows=start_line, skipinitialspace=True)
12
13    ...
14
15    # Fast Vectorized Timestamp Conversion (HH:MM:SS.ms -> HH:MM:SS.ms)
16    # Replaces the last colon with a dot to make it pandas-compatible
17    df['TimeDelta'] = pd.to_timedelta(
18        df['Timestamp'].str.replace(r':(\d+)$', r'.\1', regex=True)
19    )
20
21    # Calculate relative seconds for X-axis
22    start_t = df['TimeDelta'].iloc[0]
23    df['Seconds'] = (df['TimeDelta'] - start_t).dt.total_seconds()
24
25    return df

```

As anticipated, μ Prof CSV output is not completely standard. The first section of the file provides parameters for the following telemetry, so real parsing starts after profile records are announced.

μ Prof timestamps follow HH:MM:SS:ms, which violate ISO standards. The colon is replaced by a dot, allowing native PANDAS processing.

```

1 # Apply Smoothing (Optional)
2 if SMOOTHING_WINDOW > 1:
3     numeric_cols = df.select_dtypes(include=['float', 'int']).columns
4     cols_to_smooth = [c for c in numeric_cols if c not in ['Seconds',
5     ↪ 'RecordId']]
6     df[cols_to_smooth] = df[cols_to_smooth].rolling(window=SMOOTHING_WINDOW,
7     ↪ min_periods=1).mean()

```

A smoothing window serves the purpose of **mitigating measurement spikes that make the graph poorly-interpretable**. An algorithmical smoothing set to X means each point becomes the average of the last X seconds, revealing true macro-trends.

Our framework then provides core-level breakdown, to further validate our KUBERNETES management policies, and multi-axis visualization (Power/Temperature/Effective Frequency).

4.2. Practical effects of CPU governors in a 5G deployment

We are now interested in using such knowledge to study **the impact of varying CPU governors** in our deployment, to evaluate their ability in providing a reasonable set of behavioural shades.

4.2.1. A primer on CPU governors in Linux

As briefly introduced when deploying our ANSIBLE playbook, CPU governors in the Linux Kernel manages CPU performance through the `cpufreq` subsystem, which includes:

- **The Core:** providing the common infrastructure and interface.
- **The scaling drivers:** hardware-specific modules that know how to communicate with the CPU to act on clock speeds and voltages.

- **The scaling governors:** logical algorithms monitoring system utilization and planning targets for relevant performance parameters.

Typically, available governors include [39]:

Table 4.1: A schematic view on CPU governors

Name	Impact
performance	Bypasses dynamic scaling entirely, statically setting up frequency to the maximum value. It is the most common choice for time-constrained applications.
powersave	The inverse of performance , statically locks the CPU to the lowest available frequency.
ondemand	The traditional dynamic scaling algorithm: polls the CPU load at a defined interval. If it exceeds a defined threshold, it requests a jump to a higher frequency, until the load drops and therefore the frequency steps back down.
conservative	Designed for battery-operated devices to provide a set of gradual steps and thresholds, while introducing possible spikes for highly-demanding workloads.
schedutil	Introduced in newer Linux Kernels, deeply integrated with the Completely Fair Scheduler (CFS), leveraging load tracking at core-level.
userspace	Allows root-level user-space applications to set the CPU frequency manually.

To achieve higher frequencies, the CPU driver must also supply a higher voltage to maintain signal integrity across the silicon: a small increase in clock speed may result in a considerable increase in power consumption and heat generation.

4.2.2. The two extremes: performance vs powersave

Since we are now able to **use CPU governors as our control knobs**, we would like to evaluate the impact of varying CPU governors in a 5G deployment, focusing, in particular, on **the most consuming element: the GNB**.

As highlighted in previous chapters, the GNB will of course process incoming packets and outgoing packets, manage the shared radio channel, encode and decode relevant payloads, aggregate performance metrics, etc.

Performance governor

Let's then start from the most used governor in the realm of **real-time networking applications: the performance governor**.

We will set up a 10-minute measurement using, of course, the aforementioned AMD μ PROF timecharts in combination with our PYTHON data processing notebook.

Experimental setup

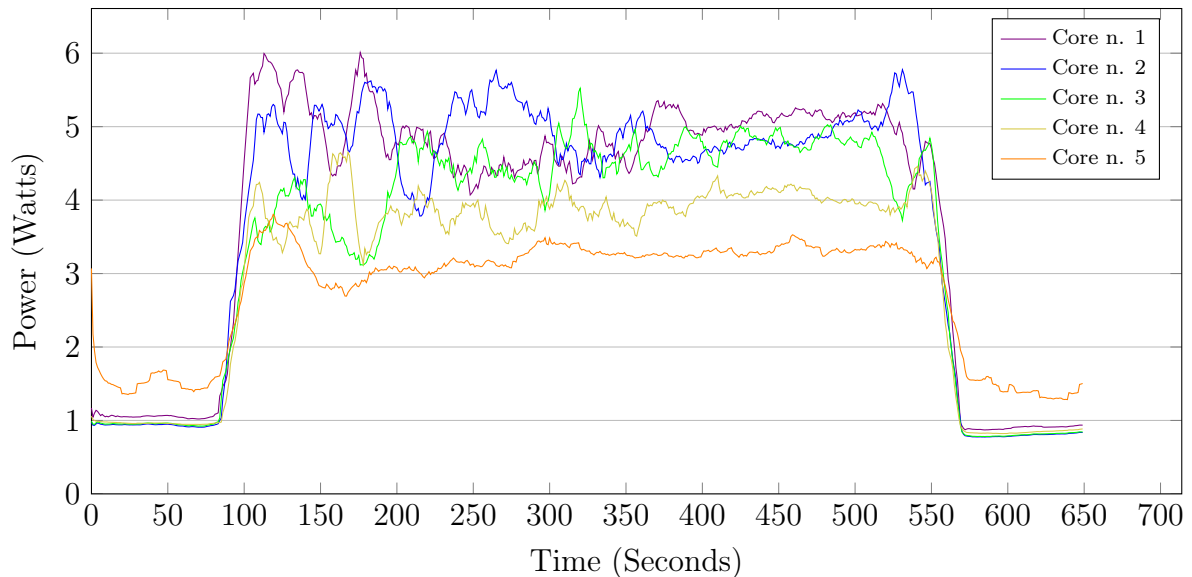
10 minutes, scripted, AMD μ PROF timechart

Core isolation and optimizations applied.

Starting in idle, installing the GNB Pod on 5 physical cores (10 logical threads).

- Connecting the first UE after 120 seconds, enabling PDU session.
- Connecting the second UE after 180 seconds, enabling PDU session.
- Connecting the third UE after 240 seconds, enabling PDU session.
- Connecting the fourth UE after 300 seconds, enabling PDU session.
- Running `iperf` speed test on all UEs after 310 seconds until session end.
- Uninstalling the GNB Pod, implicitly closing all sessions.

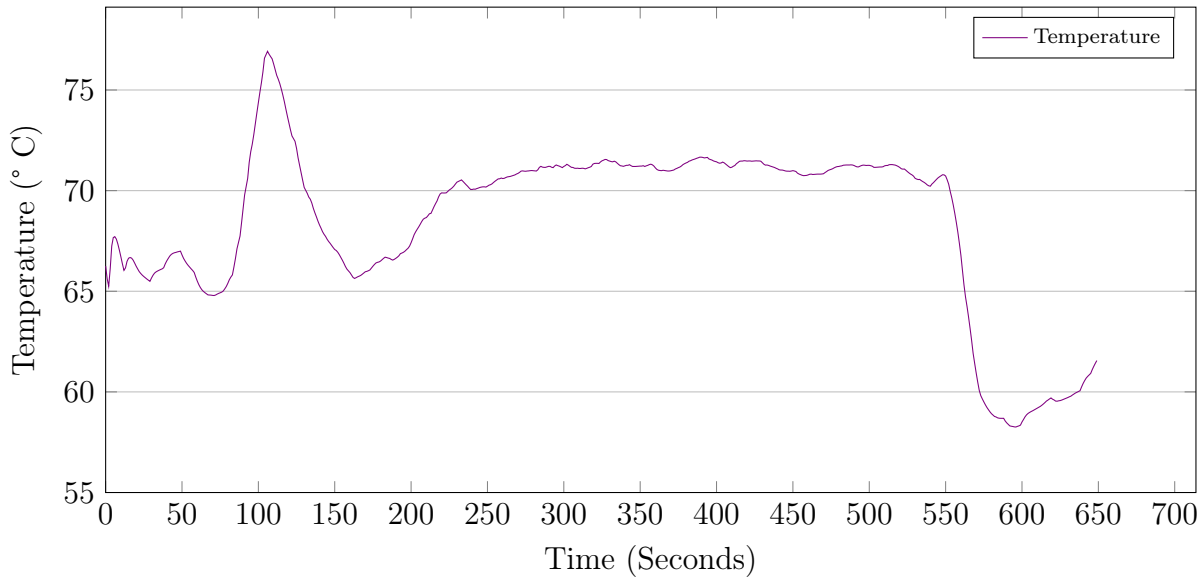
Figure 4.1: Individual core power consumption on **performance** governor



We clearly identify a **mostly-static energy impact on core power consumption**, delineating a general trend which reflects the concrete implementation we studied earlier: power consumption **scales very slowly with an increasing number of UEs and**

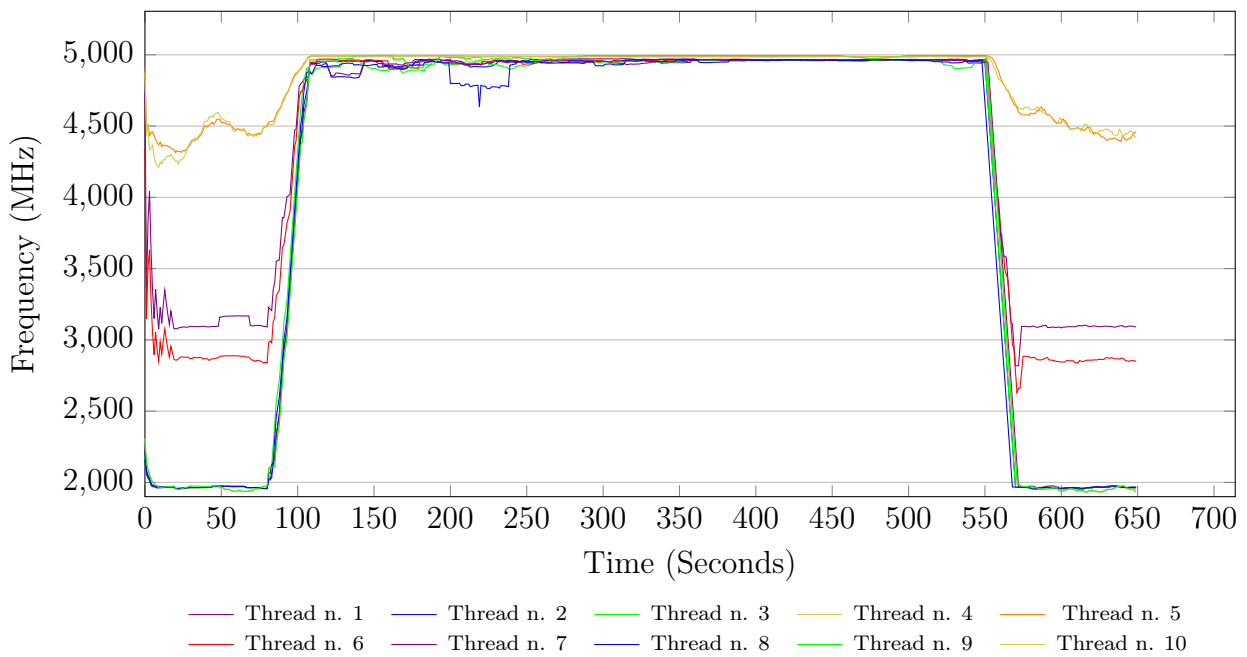
their network usage.

Figure 4.2: Temperature on performance governor



Heat dissipation confirms our intuitive analysis: when switching workloads and initiating computationally relevant networking operations (like spinning up a GNB Pod), temporary spikes in heat dissipation are to be expected.

Figure 4.3: Individual thread frequency on performance governor



The same can be said for frequency, which shows a **heavily deterministic behaviour** whenever workload is in execution.

Powersave governor

As expected, the **performance** governor exhibited an ideal behavioural pattern, which, of course, represents the best possible implementation stage in terms of raw performance.

Still, even if our deployment brings many optimizations to the table with respect to a barebone deployment, we would like to explore further and determine whether the **other extreme performance-wise** is a viable solution: let's repeat our experimental campaign adopting the **powersave** governor this time.

Experimental setup

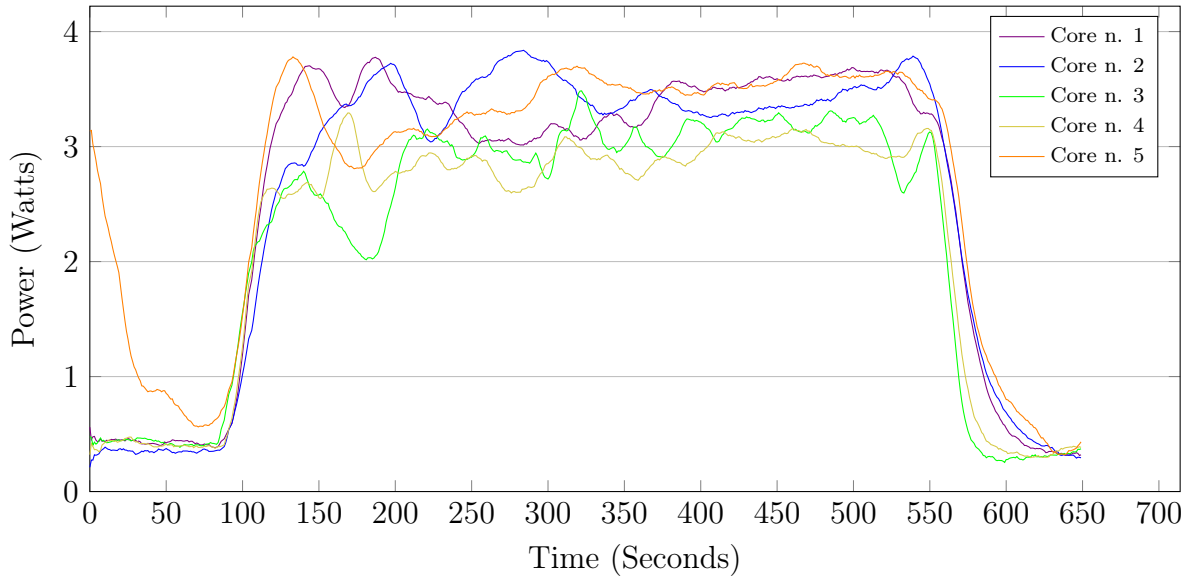
10 minutes, scripted, AMD μ PROF timechart

Core isolation and optimizations applied.

Starting in idle, installing the GNB Pod on 5 physical cores (10 logical threads).

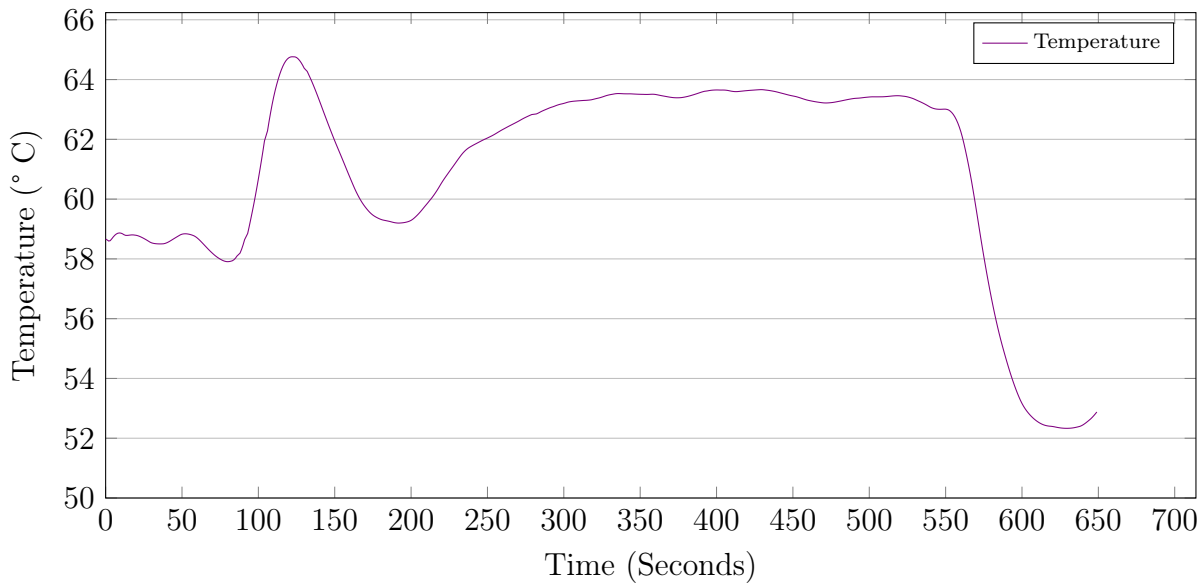
- Connecting the first UE after 120 seconds, enabling PDU session.
- Connecting the second UE after 180 seconds, enabling PDU session.
- Connecting the third UE after 240 seconds, enabling PDU session.
- Connecting the fourth UE after 300 seconds, enabling PDU session.
- Running **iperf** speed test on all UEs after 310 seconds until session end.
- Uninstalling the GNB Pod, implicitly closing all sessions.

Figure 4.4: Individual core power consumption on powersave governor

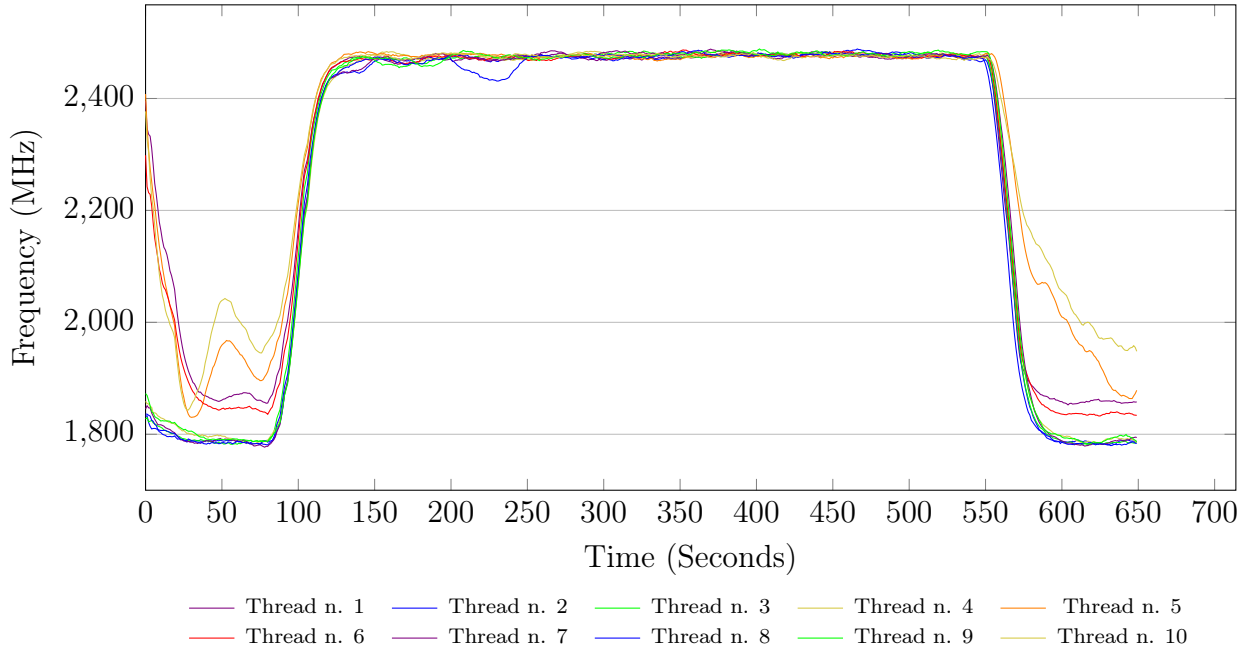


As expected, **power draw is quite impacted**: a $\sim 35\%$ average decrease can be observed.

Figure 4.5: Temperature on powersave governor



Heat dissipation draws a similar story. We also realize temperature may not be a reliable study parameter, because of its **rigidity and slow behavioural adaptation**.

Figure 4.6: Individual thread frequency on `powersave` governor

The most relevant observation can be made when analyzing clock frequencies: they are now **mostly fixed around the maximum value** provided by the Linux Kernel interfaces, which is almost half of the previous maximum value observed when using the `performance` governor.

4.3. A power governance framework for the Neutral Host management platform

Up until now, we always kept **radio throughput, interference and packet loss** out of the scope of our analysis.

In the last chapter, we built a powerful management platform for Neutral Hosts, which also allowed us to gain significant observability milestones about UEs and their metrics.

We would now like to exploit those to build a **power governance framework** to embed in our platform, to act as a vigilance solution energy-wise.

4.3.1. Escaping the Pod

Entering privileged namespaces

One of the most technically aggressive components of our solution is its relationship with **the host operating system**.

KUBERNETES PODS are, by definition, isolated via Linux **namespaces**, as we mentioned at the beginning.

Now, we need to modify hardware CPU governors and interact with Kernel modules to act on them, so we should **break this isolation**.

```

1  ...
2  kill_cmd = [ "nsenter", "-t", "1", "-m", "-u", "-i", "-n", "-p",
3              "pkill", "-f", "AMDuProfCLI" ]
4  await (await asyncio.create_subprocess_exec(*kill_cmd)).communicate()
5  ...

```

NAMESPACE ENTER (**nsenter**) is a Linux command-line tool used to run a command or program inside the namespace of another existing process.

Here, we target PID 1 (the **init** system, being **systemd** in our case) with flags **-m** (mount), **-u** (UNIX Timesharing System), **-i** (Inter-Process Communication), **-n** (network).

This effectively grants root-level host access, an absolute requirement for interacting with AMD μ PROF drivers which can only live on the physical host.

Additionally, we act on CPU governors directly via the host's mounted virtual file-system:

```

1  ...
2  def set_core_governor(core_id: int, governor: str):
3      path = f"{SYS_FS_PATH}/cpu{core_id}/cpufreq/scaling_governor"
4      try:
5          with open(path, "w") as f:
6              f.write(governor)
7  ...

```

CPU topology and Kubelet CPU management

Since we are adopting CPU core isolation, we need to establish **which cores are being currently used by a specific tenant** GNB and determine policies and consumption metrics on them:

```

1  ...

```

```

2 def get_tenant_cores(namespace: str, deployment_name: str) -> dict:
3     try:
4         pods = k8s_core_v1.list_namespaced_pod(...)
5         ...
6         pod_uid = pod.metadata.uid
7         ...
8         with open(CPU_MANAGER_STATE_FILE, "r") as f:
9             kubelet_state = json.load(f)
10
11         entries = kubelet_state.get("entries", {})
12         if pod_uid in entries:
13             container_cpu_alloc = entries[pod_uid].get("gnb", "")
14             result["all_cores"] = parse_cpuset_string(container_cpu_alloc)
15     ...

```

The KUBELET writes an internal, local state file (`cpu_manager_state`): by mapping the logical POD UID to the physical CPU-set string, the application dynamically maps logical tenants to physical CPU layouts.

The fronthaul core

In OAI, there is a specific thread responsible for **streaming samples to the USRP over the physical connection** to the host (USB/SFP+).

We therefore parse the startup logs of the GNB, cache the physical core ID of the USRP writing thread and lock it to the performance governor.

This will allow for **deterministic performance on the most time-sensitive element** of our GNB deployment.

```

1 ...
2 logs = k8s_core_v1.read_namespaced_pod_log(...)
3 match = re.search(r"trx_usrp_write_thread started on cpu (\d+)", logs)
4 if match:
5     fh_core = int(match.group(1))
6     ...
7     pod_fronthaul_cache[deployment_name] = { "uid": pod_uid, "core": fh_core }
8 ...
9 for core in tenant_cores:
10     if logged_fronthaul_core is not None and core == logged_fronthaul_core:
11         set_core_governor(core, "performance")

```

```

12     else:
13         set_core_governor(core, target_gov)

```

4.3.2. The staircase state machine

The strategically relevant algorithm in our solution resides in the evaluation of tenant performance.

We organized a loop that acts as a classic **Finite-State Automata (FSA)**, stepping through available governors:

```

1  ...
2  sample_size = max(1, int(len(tenant_kpm_keys) * 0.30))
3  sampled_keys = random.sample(tenant_kpm_keys, sample_size)
4
5  needs_scale_up = False
6  for key in sampled_keys:
7      stats = await redis_client.hgetall(key)
8      if not stats:
9          continue
10     loss_dl = float(stats.get("PacketLossRateDl_pct", 0))
11     drop_dl = float(stats.get("PdcgDropRateDl_pct", 0))
12
13     if loss_dl > PERF_THRESHOLD_LOSS_PCT or drop_dl > PERF_THRESHOLD_DROP_PCT:
14         needs_scale_up = True
15         break
16
17 if needs_scale_up:
18     current_level = min(4, current_level + 2)
19 else:
20     current_level = max(base_level, current_level - 1)
21 ...

```

- **Monte Carlo approach:** randomly sampling 30 % of the active users. If any of them exceeds 2 % packet loss rate, an alarm is triggered.
- **Asymmetric scaling (staircase):** if sampled performance degrades, the system leaps up by two governor levels to rescue connection as soon as possible. Scaling down is instead conservative (one step at a time) to prevent oscillating behaviours.

- **Hardware interrupts:** if the POD logs processing errors (e.g. overflows), UE performance evaluation is skipped and state is forced to **performance**.

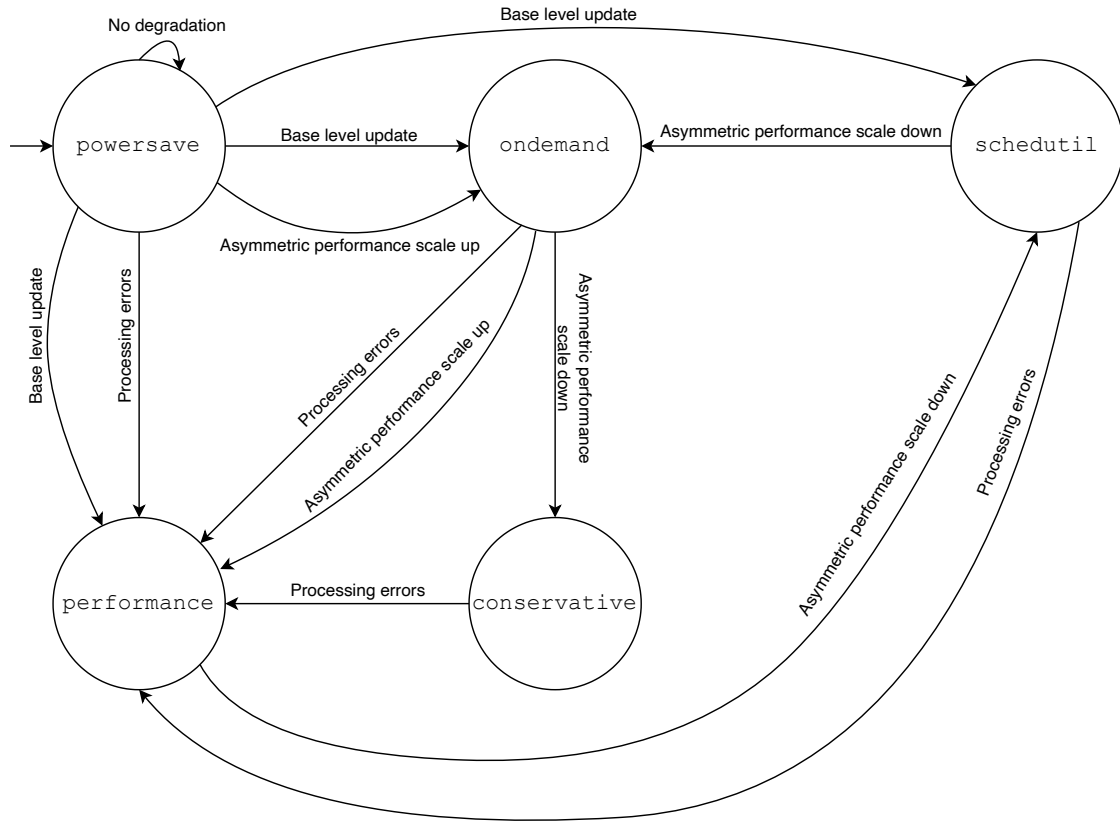


Figure 4.7: The complete Finite-State Automata describing the state evolution of the power governance framework

Note that some arcs are intentionally omitted for clarity.

4.3.3. Multi-tenant energy billing

Once μ PROF logs are parsed, we apply a custom multi-tenant cost attribution strategy:

- **Dynamic power:** the sum of Watts consumed by the specific isolated CPU cores assigned to that tenant via `Kubelet` mapping.
- **Fixed overhead:** the total socket package power — the sum of all dynamic tenant power.
- **Billing share:** the fixed overhead is shared equally across active tenants.

```

1  ...
2  fixed_power_remainder = max(0.0, total_host_power - total_dynamic_power)
3  fixed_share_per_tenant = ... fixed_power_remainder / total_active_tenants ...
4  ...
5  await conn.execute(
6      """ INSERT INTO polimi.TenantConsumption (tenant, cpu_usage, dynamic_watts,
7      fixed_watts, "start", "end") VALUES ($1, $2, $3, $4, $5, $6) """,
8      tm["plmn"], tm["cores"], tm["dyn_w"], fixed_share_per_tenant, start_ts,
9      ↪ end_ts
10 )
11 ...

```

4.3.4. Deployment and security

Standard roles are insufficient for our solution, we need to expand them to include updates of deployments (hardware resources requests and limits) and PODS logs.

```

1  ...
2  apiVersion: rbac.authorization.k8s.io/v1
3  kind: ClusterRole
4  metadata:
5      name: nh-controller-role
6  rules:
7      - apiGroups: ["apps"]
8        resources: ["deployments"]
9        verbs: ["get", "list", "patch", "update", "watch"]
10     - apiGroups: [""]
11       resources: ["pods", "pods/log"]
12       verbs: ["get", "list", "watch"]
13  ...

```

Also, we need to share the host node's PID namespace and privileges with the container (to use `nsenter` to inject commands into the host's scope).

```

1  spec:
2      serviceAccountName: nh-controller-sa
3      hostPID: true
4  ...

```

```
5     containers:
6       - name: controller
7         ...
8       securityContext:
9         privileged: true
10      volumeMounts:
11        - name: sys-fs
12          mountPath: /host-sys
13        - name: rke2-cpu-manager
14          mountPath: /kubelet-state
15          readOnly: true
16        - name: host-tmp
17          mountPath: /host-tmp
```

A note on security

From a security standpoint, this POD realizes the equivalent of a root shell on the KUBERNETES worker node.

A **Remote Code Execution (RCE) vulnerability** in API calls would yield to a meaningful compromise of the underlying host machine.

In production, API calls and the action perimeter of the POD should be further restricted to the least possible one, secured behind mutual TLS authentication and strict networking policies [26].

Since we are operating in a research-oriented environment, this should serve as a **concrete sketch of a practical and working solution**.

4.3.5. A complete experimental execution

We would finally like to make use of our complete solution, simulating **half of a normal day** operating this infrastructure as a Neutral Host.

Experimental setup

12 hours, partially scripted, AMD μ PROF timechart

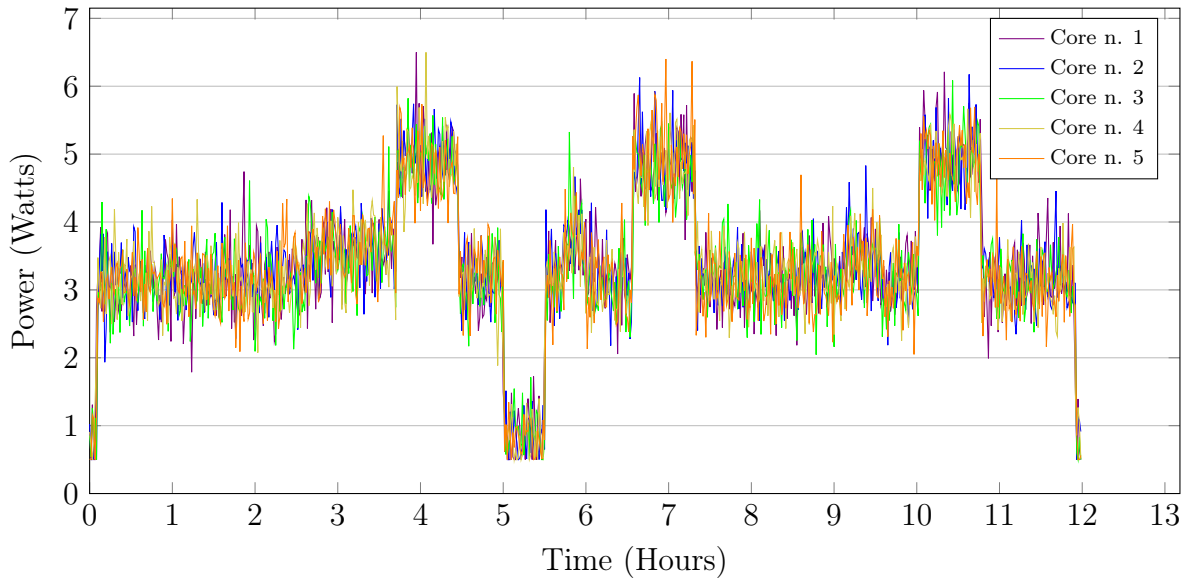
Core isolation and optimizations applied.

Starting in idle, installing the GNB Pod on 5 physical cores (10 logical threads).

- Connecting the first UE after 120 seconds, enabling PDU session.
- Connecting the second UE after 180 seconds, enabling PDU session.
- Connecting the third UE after 240 seconds, enabling PDU session.
- Connecting the fourth UE after 300 seconds, enabling PDU session.
- Running `iperf` speed test on all UEs after 310 seconds for an hour.
- Pausing speed tests for 15 minutes, **repeating until end of session.**
- Running `iperf` on all UEs for 30 minutes, **repeating until end of session.**
- Uninstalling the GNB Pod, implicitly closing all sessions.

Apart from a brief interruption after ~ 5 hours, the execution loop is quite static: 30 minutes speed tests, 15 minutes idle.

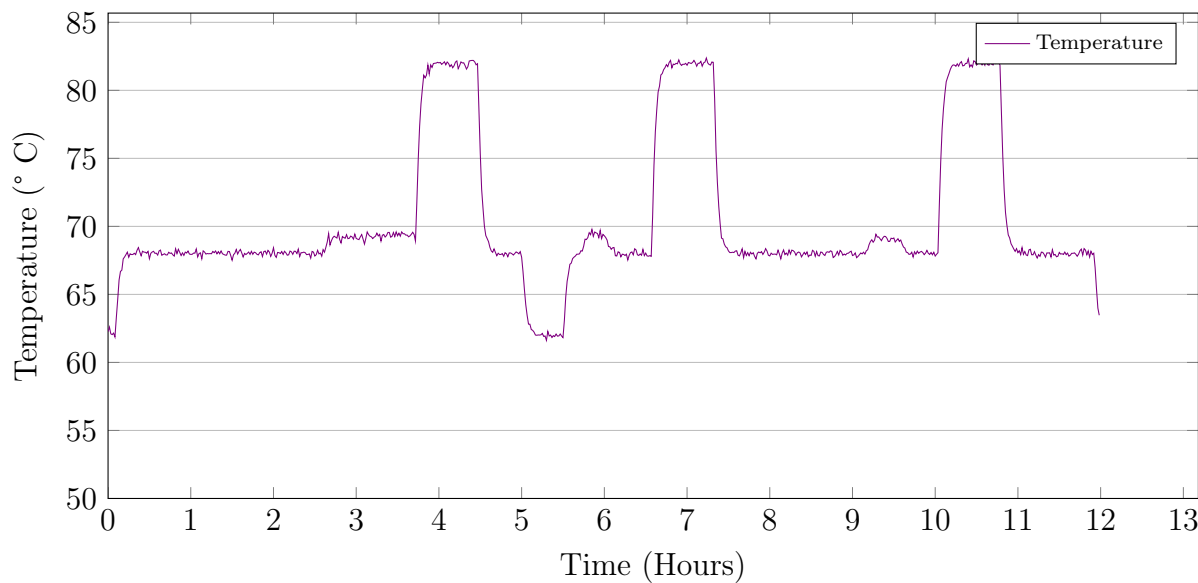
Figure 4.8: Individual core power consumption



Power draw outlines a first goal: **the GNB energy controller keeps it in powersave as much as possible**, but external interference or packet loss are always possible.

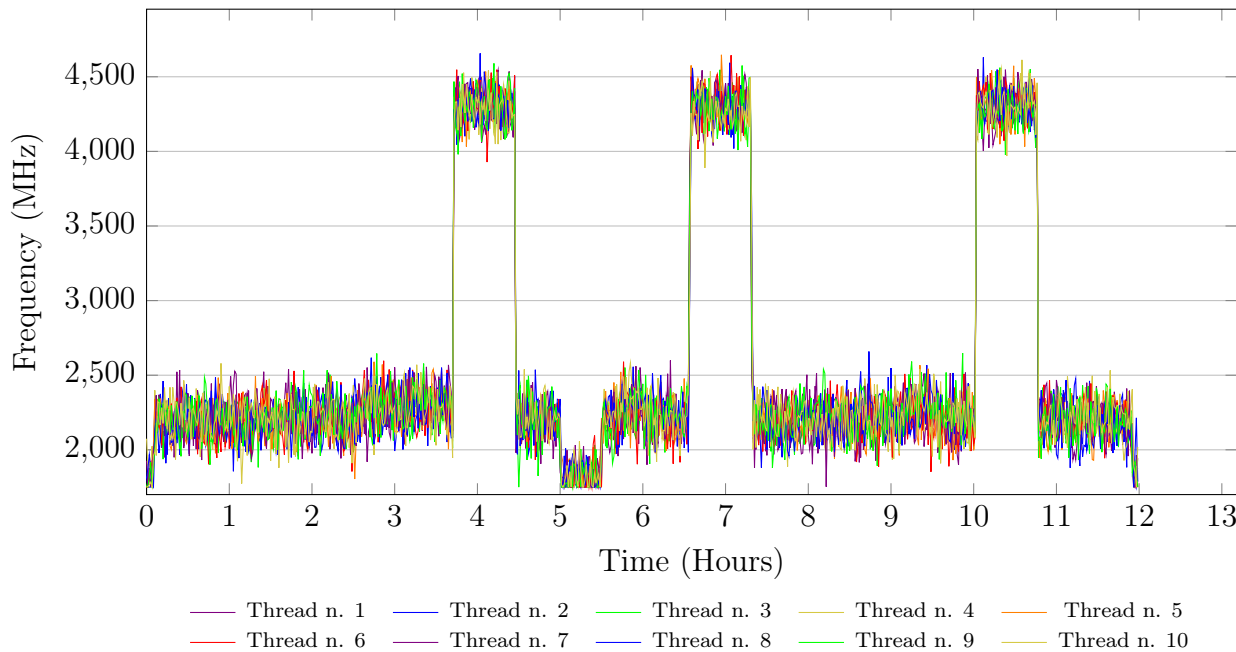
We should also mention here that, since we are using a Politecnico-licensed frequency band, it is frequently shared across laboratory deployments, so some form of interference is to be expected.

Figure 4.9: Temperature



Heat dissipation is compatible with our initial analysis and exhibits a quite **stable reference**.

Figure 4.10: Individual thread frequency



Of course, clock frequencies show the most relevant variance, clearly **outlining performance and powersave periods**.

4.3.6. End-user interface

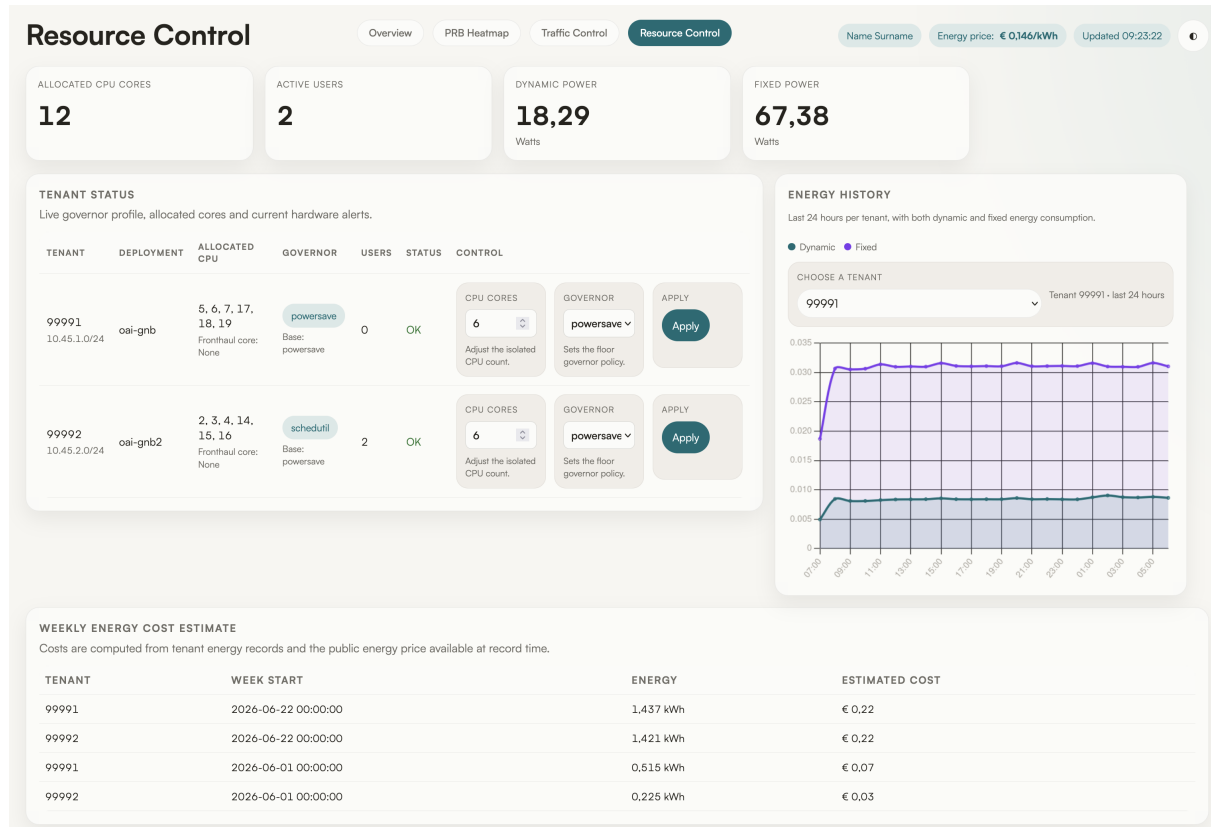


Figure 4.11: Resource control overview

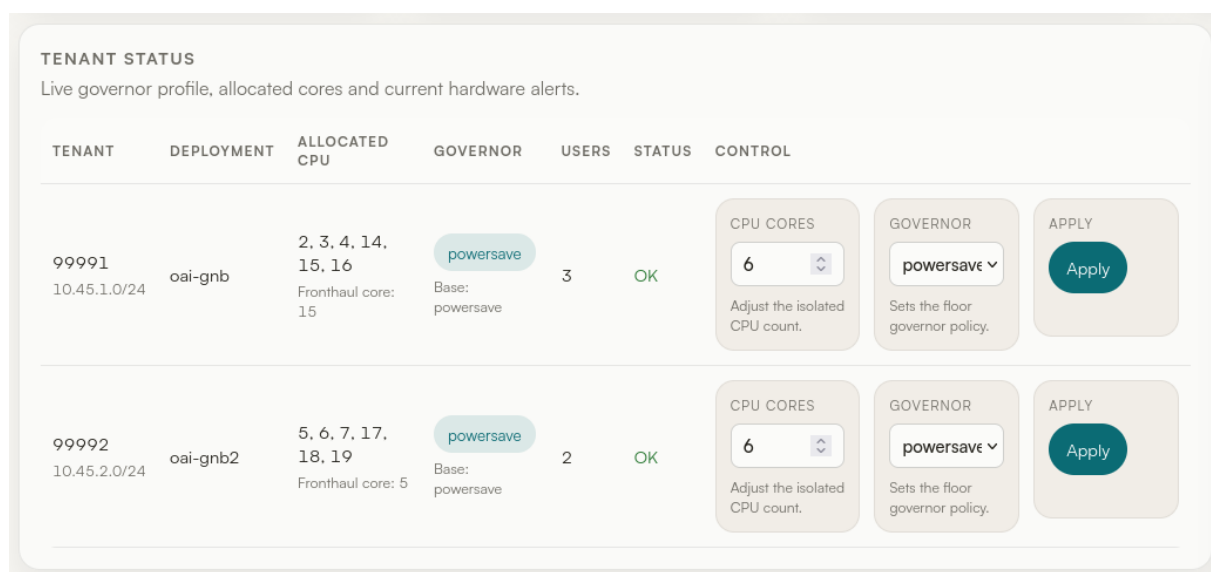


Figure 4.12: Live CPU governor floor policy and allocated threads counter

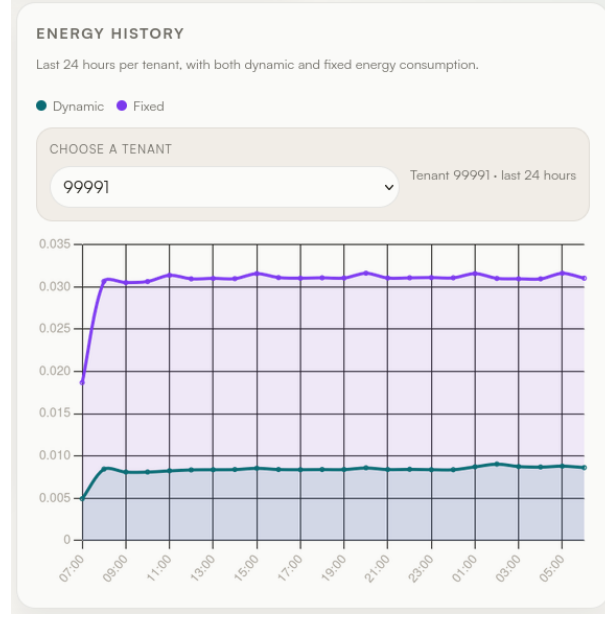


Figure 4.13: Historical power consumption metrics, on a per-tenant basis

4.4. Use-cases

4.4.1. User and tenant billing

As anticipated, our tenant power management solution executes a 60-second sliding window telemetry capture strategy, detailing power draw of individual cores and tenants, alongside the global package power draw.

The final billing algorithm splits them into dynamic and fixed costs:

- **Dynamic power** (P_D): the sum of the Watts consumed exclusively by the isolated physical cores assigned to a single tenant.
- **Total dynamic power** (P_{D_TOT}): the sum of all P_D across all currently active tenants.
- **Fixed overhead** (P_{F_TOT}): the result of $P_{SOCKET} - P_{D_TOT}$.

This is symbolically represented by the energy needed to run the Core Network and management infrastructure as a whole.

Refinements in fixed costs establishment are definitely possible and should be considered in commercial implementations.

- **Tenant share** (P_F): the final billed wattage for each tenant is $P_D + \frac{P_{F_TOT}}{N_{TENANTS}}$.

We are also keeping a detailed **track of each user session, recording, for each of them, the initialization and termination timestamps.**

For each recorded session, we keep track of the precise number of transmitted and received bytes: integrating this information with relevant details about billing cycles could create a perfect fit for a customer billing management solution.

For each billing cycle C , whose start instant is defined as C_S and the end one is defined as C_E , with a fixed rate R_{TX} for each transmitted byte and R_{RX} for each received one, For each UE session S with initial timestamp S_S and the end one S_E , S_{TX} the number of transmitted bytes and S_{RX} the number of received ones, we define the resulting user billing R_U :

$$R_U = R_{RX} \cdot S_{RX} + R_{TX} \cdot S_{TX} \quad \forall S \mid C_S < S_S \wedge C_E > S_E$$

The underlying assumption is that every session is interrupted (either logically or practically) right before the beginning of a new billing cycle (and a new one is therefore set up).

4.4.2. Real-time performance and SLA monitoring

Mobile Network Operators sign strict SLAs with Neutral Host providers, mandating, for example, a certain availability and uptime metric for their infrastructure.

We built an intrinsically scalable deployment by using KUBERNETES, one of the main enablers for distributed systems deployments and ANSIBLE, which, as we briefly commented at the beginning, is able to apply and enforce a PLAYBOOK actions to an indefinite amount of machines.

Horizontal scalability is therefore a trivial exercise to reach Service Level Agreements (SLA).

Through our Neutral Host management platform, we are also adding:

- **Radio signal quality:** RSRP and SNR monitoring, constantly updated and pushed from the GNB to the platform's dashboard.
- **Current throughput:** measured directly from Key Performance Metrics extracted from our customized XAPP running on the RAN Intelligent Controller, also constantly updated and pushed to the platform's dashboard.
- **Physical Resource Blocks:** a schematic view of currently used PRBs is also extracted from the RIC and pushed to the real-time PRB heatmap, showing, in detail, each user and their current allocation for PRBs.

- **Traffic control:** thanks to the application of traffic control policies, we are able to investigate and limit one or more UEs.

As a response to SLA violation, our power management framework acts immediately by pushing a forced **governor** update thanks to the staircase-inspired state machine algorithm, to mitigate the effects of temporary packet loss or signal integrity degradation.

4.4.3. Performance correlation

In distributed telecommunications solutions, identifying the root cause for performance degradation is notoriously difficult.

Our platform acts as a correlation engine, by providing extensive logging and metrics aggregation insights exploiting privileged access to the concrete infrastructure implementation.

As we have just commented, we are now able to **detect hardware errors, overflows, interference, samples decode and encode issues and packet loss rates across all UEs and gNBs**, drastically reducing the Mean Time To Resolution (MTTR).

4.4.4. Anomaly and abuse detection

In a shared radio environment, the Medium Access Control (MAC) layer scheduler is responsible for assigning PRBs to UEs.

While proportionally fair algorithms attempt to balance throughput and fairness, certain users (or compromised ones, think for example at vulnerable IoT devices participating in a botnet) can **monopolize cell bandwidth, starving other users on the same tenant**.

Traditional mobile networks use Deep Packet Inspection to find abusers. In 5G, user plane traffic is encapsulated in tunnels and often encrypted.

Abuse detection must therefore happen at the Radio Link Control (RLC) or MAC layers using properly-sized telemetry.

Since our solution maps the temporary RNTI back to the permanent IMSI and IPv4 address via the customized mapping daemon, identifying a monopolizing user quite simple: **KPMs indicate clearly the users consuming an unusual amount of available resources** and packet loss metrics are also available for precise abuse analysis.

4.4.5. Spatial user mapping and positioning

One of the most revolutionary aspects of recent 5G releases is the native support for high-accuracy positioning.

As explored in recent literature, precision location awareness unlocks a wide range of use-cases, from automated intersection crossing to precise indoor logistics.

Let's first explore **classic localization systems for wireless networks, which can be readily applied** to our solution.

Classic localization in wireless networks

In our case, the most interesting localization systems are model-based ones using power-based parameters (e.g. RECEIVED SIGNAL STRENGTH, RSS).

Triangulation:

The target estimates distances $d_1, d_2, \dots, d_n$ from n anchor nodes in positions $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.

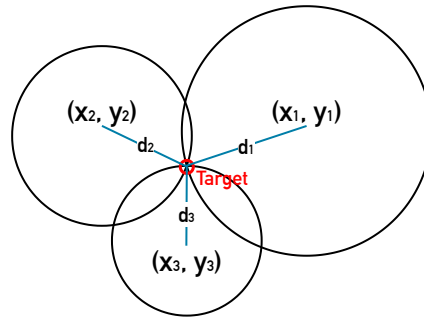


Figure 4.14: A basic triangulation setting

Target position is the intersection of the (at least three, of course) circumferences having $d_1, d_2, \dots, d_n$ as radius and $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ as their centers.

Fingerprint-based localization:

The FINGERPRINTING approach consists of two basic steps:

1. Construct a radio map (FINGERPRINT DATABASE) of the environment, storing for each position (x, y) the RSS from all visible monitoring points $(s_1, s_2, \dots, s_n)$.
2. When the target needs to be localized, transmit the RSS from monitoring points to the database and get the position with the most similar fingerprint.

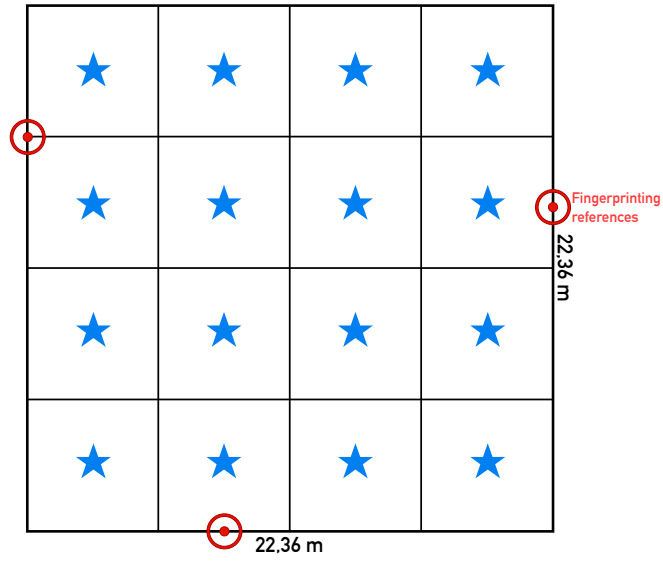


Figure 4.15: A sketched example of a FINGERPRINTING DATABASE construction on a sample underground field

Since our solution already provides relevant radio parameter estimation, a sketched implementation **can be worked out in both private and controlled deployments.**

5 | Conclusion

In the modern era, Mobile Network Operators play a fundamental role. In the increasingly complex landscape we live in, external pressures are driving the development of paradigmatical shifts towards **transparent, efficient and more inspectable systems**.

As MNOs move towards open-source, virtualized and disaggregated architectures like O-RAN, the challenges for Neutral Hosts of exposing an observability layer to their users requires careful architectural planning.

While doing so, we should not forget about the importance of **power and operational efficiency**, optimizing for the increasingly-higher demand for mobile and connected experiences.

This work challenged these needs by proposing, implementing and validating an efficient and observable platform for multi-tenant Neutral Host deployments, composed of:

- **A Neutral Host deployment:** We successfully built and deployed a multi-tenant Neutral Host setup on a KUBERNETES cluster. The architecture and its implementation details, described in Chapter 2, serves as a reference point for all kinds of practical implementations, spanning from centralized, single-node clusters to complex multi-node and geographically distributed setups.
- **A custom and efficient Kubernetes distribution:** As a response to the underlying incompatibilities currently present in an OKD setup, we replicated its main advantages through ANSIBLE, an Infrastructure as Code solution. Through our custom solution, we have been able to deploy general primitives aiming at profiling, tuning and managing hardware, system and networking resources.
- **An open-source 5G stack:** We prepared the ground for our platform by deeply analyzing and customizing the industry-leading standards for open-source, research-oriented 5G deployments.
- **An observability platform for Neutral Hosts:** We leveraged such knowledge to exploit the privileged access to the newly-developed infrastructure as a whole to

close the visibility gap in existing Neutral Host deployments. Neutral Host users are now able to inspect and deeply control the usage of hardware resources for their tenants, throughput, performance and radio metrics for their UEs and enforce traffic control limitations on each of them.

- **A power governance framework:** We finally proposed a power governance solution as an integration to the Neutral Host management platform, which allowed us to validate the chosen implementation strategies. The resulting solution is readily applicable to a wide variety of use-cases, such as real-time performance and SLA monitoring, anomaly and abuse detection, users and tenants billing cycles.

5.1. Future Works

5.1.1. Positioning Reference Signals and Digital Twin paradigm

Positioning Reference Signals in 5G

3GPP Release 16 introduced in the standard enhanced Downlink and Uplink Positioning Reference Signals (PRS) [18].

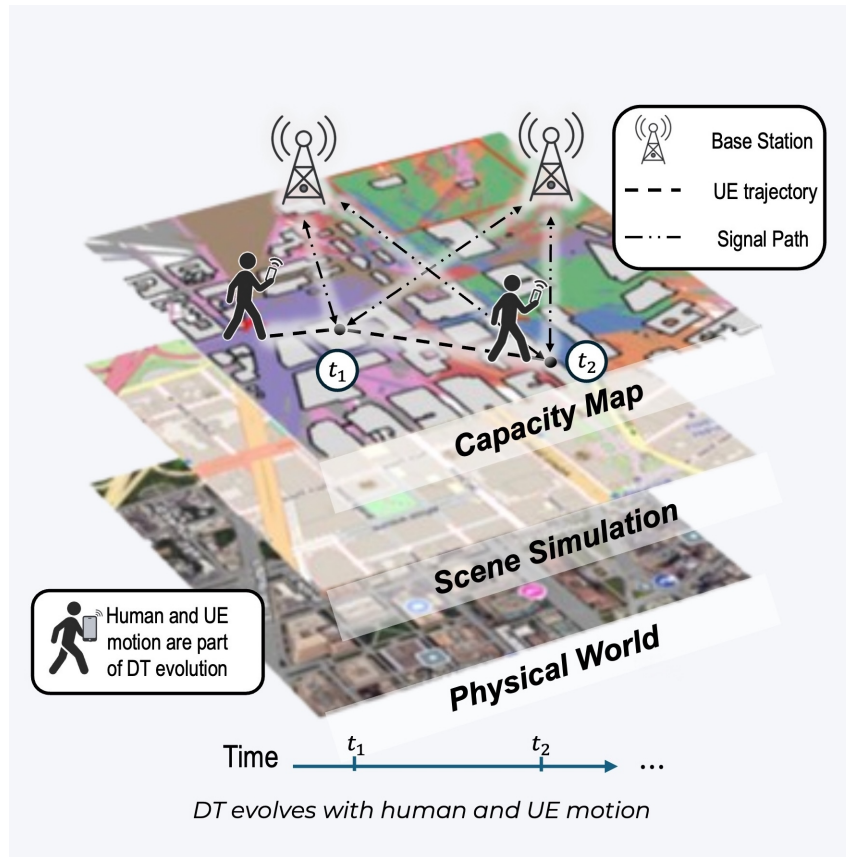
By making use of larger bandwidths, the time-domain sampling resolution drastically increases, allowing the system to distinguish multipath reflections down to fractions of seconds.

These new reference signals **improve positioning accuracy and lower the communication overhead**.

The Location Management Function (LMF) orchestrates the positioning procedures within the 5G Core Network.

A key feature of PRSs is their ability to **report metrics from multiple gNBs simultaneously**: multiple base stations can transmit PRSs in a coordinated setting using specific radio parameters.

Such knowledge enables novel capabilities, such as **Spatial SLA Mapping**: the system can plot KPM degradation on a physical map, deriving physical obstacles in correspondence of which packet loss estimations spikes (causing therefore radio performance degradation).



The Digital Twin paradigm

In the context of telecommunications, the Digital Twin paradigm represents a significant leap in network management, focusing, in particular on **dynamic resource allocation and scaling** [27].

A Digital Twin allows for **proactive and predictive management**: simulating changes in the Virtual Space and observe their impact on performance metrics.

In the 5G context, the Virtual Space must model both the hardware and the logical state of UEs, the radio environment and energy consumption profiles.

For a Digital Twin to be accurate, its network must use real-time and comprehensive metrics.

The GNB-CORE mapper we described earlier is a first step towards this goal: it ultimately creates a **real-time state database of the system**, by resolving the fragmentation of 5G identifiers.

Similarly, the power governance framework offers an in-depth view of the current state of the underlying physical hardware.

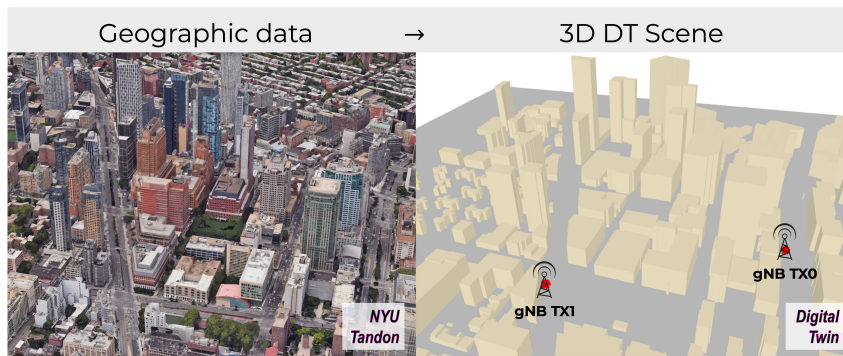
The hierarchical and interconnected nature of the resulting deployment makes it suitable

for common graph representation, allowing for easy performance impact tracking across UEs and hardware resources.

We now therefore have the relevant context to train specialized Machine Learning models: simulating a real-world scenario (e.g. the deployment of a new tenant on the current infrastructure) is now easier.

We are also able to collect a complete picture now: the Digital Twin, thanks to the Positioning Reference Signals, is able to construct **a 3D environment thanks to Computer Graphics techniques**.

With such model, imminent drops in signal quality and interference metrics can be accurately predicted and perform proactive mobility procedures (e.g. handover to a different Radio Unit).



5.1.2. Official integration with open-source solutions

Currently, the Neutral Host management platform does not modify the underlying implementation of OPEN5GS and OPENAIRINTERFACE: we wanted to make our solution as general as possible, and we felt that creating a custom spin of a widely known and frequently updated open-source project goes against this principle.

An official integration to OAI and OPEN5GS would enable a wide range of possibilities, in an easier and more stable setting:

- **Identity mapping:** relying on the intersection of GNB, AMF and SMF logs is an unstable solution.
Integrating a common data structure for fixed IMSI-NGAP ID matching would strip down many complexities in our solution.
- **Traffic control:** since, as mentioned, the current implementation of pure radio slicing (e.g. PRB blocklisting) on USRPs is undergoing several operational complexities [7], the best choice for our solution has been adopting Linux-native mechanisms for traffic policy enforcement.

Re-using the same mechanisms employed by [8] would allow for pure radio slicing and enable further control on UEs performance and throughput.

- **Hardware resource management:** we currently rely on KUBERNETES-native mechanisms for POD-patching and resource usage updates. Official solutions from either OAI or SRSRAN would allow for easier and more integrated management of hardware resources allowing for straightforward consolidation of chosen optimization strategies, without impacting on availability or Service Level Agreements.

5.1.3. Machine Learning for proactive control

The power governance framework we implemented in chapter 4 **reacts to the current conditions being detected in real-time by our customized containerized environment.**

This includes:

- **Monte Carlo approach:** randomly sampling 30 % of the active users. If any of them exceeds 2 % packet loss rate, an alarm is triggered.
- **Asymmetric scaling (staircase):** if sampled performance degrades, the system leaps up by two governor levels to rescue connection as soon as possible. Scaling down is instead conservative (one step at a time) to prevent oscillating behaviours.
- **Hardware interrupts:** if the POD logs processing errors (e.g. overflows), UE performance evaluation is skipped and state is forced to **performance**.

In our considerations, we ignored seasonal or recurrent deviations from the most common operational environment: this includes highly-populated environments or enterprise areas, being characterized by traffic spikes in recurrent and predictable time frames.

On the other hand, tenants may want to optimize their energetic footprint by analyzing energy rates and operating context to reduce their hardware resource requests when adequate.

5.1.4. Adapting to novel hardware platforms: ARM/RISC-V

For decades, x86 architectures have dominated the datacenter scene. Processors from INTEL and AMD formed the bedrock of enterprise computing, cloud and telecommunications infrastructure.

We are now facing the increasing development of two Instruction Set Architectures (ISAs): ARM and RISC-V.

Many optimizations on INTEL have been already explored in the 5G Neutral Host deployment space (e.g. [4]). We extended such work to AMD platforms, keeping the most relevant and impactful landmarks, while adopting the widest set of generally-available and open-source technologies.

ARM and RISC-V deployments in the mobile radio networks space is a mostly unexplored direction and would take great advantage of the mechanisms shown by this work.

Bibliography

- [1] Dharmendra Ahuja and Varshith Reddy. An Overview of OpenShift Architecture and Enterprise Container Platform Management. *Journal of Science Engineering Technology and Management Sciences*, 1:224–232, 12 2024.
- [2] Yazan M. Allawi, Alaelddin F. Y. Mohammed, Joohyung Lee, and Seong Gon Choi. A Sustainable Business Model for a Neutral Host Supporting 5G and beyond (5GB) Ultra-Dense Networks: Challenges, Directions, and Architecture. *Sensors*, 22(14), 2022.
- [3] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.
- [4] Mattia Bevilacqua. EcoRAN: an energy-aware optimization framework for multi-tenant neutral host O-RAN deployments. Master’s thesis, Politecnico di Milano, <https://hdl.handle.net/10589/240867>, 7 2025.
- [5] Leonardo Bonati, Michele Polese, Salvatore D’Oro, Stefano Basagni, and Tommaso Melodia. NeutRAN: An Open RAN Neutral Host Architecture for Zero-Touch RAN and Spectrum Sharing. *IEEE Transactions on Mobile Computing*, 23(5):5786–5798, 2024.
- [6] Sebastian Böhm and Guido Wirtz. Immutable Operating Systems: A Survey. 05 2023.
- [7] Chieh-Chun Chen. Add 5G MAC slice with NVS algo and RC ran func. https://gitlab.eurecom.fr/oai/openairinterface5g/-/merge_requests/24581, 2023.
- [8] Hai Cheng, Salvatore D’Oro, Rajeev Gangula, Sakthivel Velumani, Davide Villa, Leonardo Bonati, Michele Polese, Tommaso Melodia, Gabriel Arrobo, and Christian Maciocco. ORANSlice: An Open Source 5G Network Slicing Platform for O-RAN. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom ’24, page 2297–2302, New York, NY, USA, 2024. Association for Computing Machinery.

- [9] OKD Contributors and Red Hat. Create a Single Node OKD (SNO) Cluster with Assisted Installer. <https://okd.io/docs/project/guides/sno-assisted-installer/>, 2026.
- [10] OKD Contributors and Red Hat. Installing OpenShift on a single node. https://docs.okd.io/latest/installing/installing_sno/install-sno-installing-sno.html, 2026.
- [11] The Flatcar Project Contributors. Flatcar Container Linux: documentation. <https://www.flatcar.org/docs/latest>, 2026.
- [12] Intel Corporation Software Defined Datacenter Solutions Group. Multiple Network Interfaces in Kubernetes. <https://builders.intel.com/docs/networkbuilders/multiple-network-interfaces-in-kubernetes-application-note.pdf>, 2018.
- [13] Supraja Gurijala. Towards Zero-Drift Kubernetes Infrastructure: A Unified Model Using OSTree and Immutable Linux Distributions . *Sarcouncil Journal of Engineering and Computer Sciences*, 4(8), August 2025.
- [14] Red Hat. OpenShift Container Platform 4.21: Architecture. https://docs.redhat.com/en/documentation/openshift_container_platform/4.21/pdf/architecture/OpenShift_Container_Platform-4.21-Architecture-en-US.pdf, 2026.
- [15] Antoine Hitayezu, Jui-Tang Wang, and Tang Chen Chia. From 2G to 6G: A comprehensive survey on the evolution and enhancement of AKA protocols for next-generation wireless networks. *ICT Express*, 12(3):630–640, 2026.
- [16] Advanced Micro Devices Inc. AMD μ Prof: user guide. <https://docs.amd.com/go/en-US/57368-uProf-user-guide>, 2026.
- [17] Advanced Micro Devices Inc. EPYC™ System Management Interface (E-SMI) In-band Library. https://github.com/amd/esmi_ib_library, 2026.
- [18] Lorenzo Italiano, Bernardo Camajori Tedeschini, Mattia Brambilla, Huiping Huang, Monica Nicoli, and Henk Wymeersch. A Tutorial on 5G Positioning. *IEEE Communications Surveys and Tutorials*, 27(3):1488–1535, 2025.
- [19] Praveen Chaitanya Jakku. Advanced Ansible Playbook Best Practices: Why Infrastructure Automation Became Critical for DevOps. *International Journal on Science and Technology*, 11, 08 2020.

- [20] The kernel development community. AMD HSMP interface. https://docs.kernel.org/arch/x86/amd_hsmp.html.
- [21] Heiko Koziolk and Nafise Eskandani. Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23, page 17–29, New York, NY, USA, 2023. Association for Computing Machinery.
- [22] Sidero Labs. Talos Linux: documentation. <https://docs.siderolabs.com/talos/latest/overview/what-is-talos>, 2026.
- [23] Gabriel Lando, Lucas Augusto Fonseca Schierholt, Mateus Paludo Milesi, and Julian Araujo Wickboldt. Evaluating the performance of open source software implementations of the 5G network core. In *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7, 2023.
- [24] The Linux Foundation LSB Workgroup. Filesystem Hierarchy Standard. https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf, 2015.
- [25] Gabriel Jorge Menezes. Unofficial derivation for AMD μ Prof for NixOS. <https://github.com/Gab-Menezes/nix-amd-uprof>, 2024.
- [26] Zlatan Morić, Vedran Dakić, and Tomislav Čavala. Security Hardening and Compliance Assessment of Kubernetes Control Plane and Workloads. *Journal of Cybersecurity and Privacy*, 5(2), 2025.
- [27] Huan X. Nguyen, Ramona Trestian, Duc To, and Mallik Tatipamula. Digital Twin for 5G and Beyond. *IEEE Communications Magazine*, 59(2):10–15, 2021.
- [28] The Vector project. Vector Remap Language (VRL): A domain-specific language for modifying your observability data. <https://vector.dev/docs/reference/vrl/>, 2026.
- [29] Guillaume Raffin and Denis Trystram. Dissecting the software-based measurement of CPU energy consumption: a comparative analysis, 2024.
- [30] Petra Raussi, Heli Kokkonen-Tarkkanen, Kimmo Ahola, Antti Heikkinen, and Mikko Uitto. Prioritizing protection communication in a 5G slice: Evaluating HTB traffic shaping and UL bitrate adaptation for enhanced reliability. *The Journal of Engineering*, 2023(9):e12309, 2023.
- [31] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The Real-Time

- Linux Kernel: A Survey on PREEMPT_RT. *ACM Comput. Surv.*, 52(1), February 2019.
- [32] Damir Regvar, Ivan Vlahović, and Mislav Balković. A Controlled Comparative Evaluation of Infrastructure as Code Tools: Deployment Performance and Maintainability Across Terraform, Pulumi, and AWS CloudFormation. *Applied Sciences*, 16(6), 2026.
- [33] Robert Schmidt, Mikel Irazabal, and Navid Nikaein. FlexRIC: an SDK for next-generation SD-RANs. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 411–425, New York, NY, USA, 2021. Association for Computing Machinery.
- [34] Robert Schöne, Thomas Ilsche, Mario Bielert, Markus Velten, Markus Schmidl, and Daniel Hackenberg. Energy Efficiency Aspects of the AMD Zen 2 Architecture. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 562–571, 2021.
- [35] srsRAN Project. O-RAN CU-DU Split: tutorial. https://docs.srsran.com/projects/project/en/latest/tutorials/source/cu_du_split/source/index.html, 2026.
- [36] srsRAN Project. srsRAN gNB with COTS UEs: tutorial. <https://docs.srsran.com/projects/project/en/latest/tutorials/source/cotsUE/source/index.html>, 2026.
- [37] Mohamad Wani, Mathias Kretschmer, Bernd Schröder, Andreas Grebe, and Michael Rademacher. Open RAN: A Concise Overview. *IEEE Open Journal of the Communications Society*, 6:13–28, 2025.
- [38] Sierra Wireless. Mobile Broadband Package for Linux. <https://source.sierrawireless.com/resources/airprime/software/mbpl/mbpl-software-latest/>, 2026.
- [39] Rafael J. Wysocki. CPU Performance Scaling. <https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html>, 2017.
- [40] Jiaxun Yang. RyzenAdj: Adjust power management settings for Ryzen Mobile Processors. <https://github.com/flygoat/ryzenadj>, 2026.
- [41] Hexuan Yu, Md Mohaimin Al Barat, Yang Xiao, Y. Thomas Hou, and Wenjing Lou. Closing the Visibility Gap: A Monitoring Framework for Verifiable Open RAN Operations, 2025.

A | Appendix A

A.1. Code publishing notes

All the code developed throughout this work has been made publicly available on GITHUB and BITBUCKET, via the following URLs:

GITHUB: <https://github.com/andreacaravano/neutral-host-management-platform>

BITBUCKET: <https://bitbucket.org/andreacaravano/neutral-host-management-platform>

All artifacts have some deployment and general notes attached, to ease the inspection procedures.

A.2. Live simulative demo

A live demo that makes use of a simulative environment, heavily inspired from our real-world deployment, is available on the author's website:

LIVE SIMULATIVE DEMO: <https://tesi.andreacaravano.net>

The demo runs on a 30-minute cycle in which radio and performance metrics are generated from a random pool of meaningfully constructed operational models, that keep track of both signal quality and raw resulting throughput.

List of Figures

1.1	A schematic view of the old 4G RAN architecture, which shows the broad usage of proprietary interfaces	9
1.2	A brief overview at RAN disaggregation in O-RAN, outlining standardized interfaces	11
1.3	The construction of a system image	20
2.1	A summary of ANSIBLE's architectural model	36
3.1	SYSMOCOM SYSMOISIM-SJA2 programmable SIM cards	66
3.2	HID OMNIKEY 6121 Smart Card Reader/Writer	67
3.3	A commercial USB 3.0 setup for UE testing	68
3.4	A first architectural overview outlining the presence of the TRAFFIC CONTROLLER	76
3.5	The resulting architecture, where the use of VECTOR and FLEXRIC is highlighted	91
3.6	A partial infrastructural overview showing the GNB-CORE mapper in action	93
3.7	A complete architectural overview including all the components we developed and integrated in our solution	96
3.8	E-R model diagram for the Neutral Host management platform	97
3.9	Key-value pairs stored in REDIS	99
3.10	Tenant Live overview	100
3.11	Historical performance and radio usage metrics on a per-UE basis	101
3.12	Live PRB usage heatmap for tenant n. 1	101
3.13	Live PRB usage heatmap for tenant n. 2	102
3.14	Live traffic overview	103
3.15	On each currently active session, we can enforce custom rate limiting strategies	103
4.1	Individual core power consumption on performance governor	111
4.2	Temperature on performance governor	112

4.3	Individual thread frequency on performance governor	112
4.4	Individual core power consumption on powersave governor	114
4.5	Temperature on powersave governor	114
4.6	Individual thread frequency on powersave governor	115
4.7	The complete Finite-State Automata describing the state evolution of the power governance framework	119
4.8	Individual core power consumption	122
4.9	Temperature	123
4.10	Individual thread frequency	123
4.11	Resource control overview	124
4.12	Live CPU governor floor policy and allocated threads counter	124
4.13	Historical power consumption metrics, on a per-tenant basis	125
4.14	A basic triangulation setting	128
4.15	A sketched example of a FINGERPRINTING DATABASE construction on a sample underground field	129

Acknowledgements

Ringrazio sentitamente tutte le persone che mi hanno supportato in questo percorso, primi tra tutti i miei genitori e la mia famiglia, senza il cui sostegno continuo non avrei potuto raggiungere questo importante obiettivo.

Un ringraziamento particolare va al mio relatore, Prof. Marco Mezzavilla, per l'incoraggiante supporto e guida durante tutto il lavoro di tesi.

Non si può inoltre dimenticare il competente contributo dei Prof. Antonio Capone ed Eugenio Moro, che hanno saputo fornire una direzione molto precisa ed altamente formativa al progetto e per la quale non posso che esserne riconoscente.

Preziosissimi anche i contributi di Davide, Alessandro e Daniele del laboratorio RESTART presso il DEIB.

Una doverosa menzione va inoltre alla comunità open-source, sia in ambito 5G e reti radio mobili/telecomunicazioni che non. Una parte non indifferente del nostro pianeta si fonda sulle tecnologie e sulle implementazioni che svariati sviluppatori molto competenti portano avanti, spesso come progetti secondari o nel loro tempo libero, talvolta senza alcun ritorno economico.

L'attuale contesto sta finalmente risvegliando una maggior consapevolezza di questi aspetti, che tuttavia non andrebbero mai trascurati e dimenticati.

Un grazie speciale va infine a tutti i compagni di corso e amici che mi hanno accompagnato durante il percorso triennale e magistrale: i momenti di confronto, leggerezza e comprensione reciproca hanno rappresentato momenti fondamentali del percorso.

