# Evaluation of vTAP and Monitoring alternatives for virtualized 5G Environments

Master's thesis to obtain the academic degree
*Master of Science*
in the degree programme Master of Communication Systems and Networks
at the Faculty of Information, Media and Electrical Engineering
of the University of Applied Sciences of Cologne


| | |
|---|---|
| Submitted by: | Edisson Andres Zurita Hidalgo |
| Matriculation-Nr.: | 11156096 |
| Address: | Betzdorfer Str. 2 |
| | 50679 Köln |
| | edisson_andres.zurita_hidalgo@smail.th-koeln.de |

| | |
|---|---|
| Set up by: | Prof. Dr. Andreas Grebe |
| Second reviewer: | Prof. Dr. René Wörzberger |


Cologne, 10.06.2024

# Abstract

In the last two decades, the telecommunication industry has experienced a significant development in every possible area, beginning with a severe need to transition from IPv4 to IPv6 addressing system due to multiple reasons: the upcoming need for higher bandwidth which triggered the irruption of fifth-generation (5G) mobile networks, the spread of internet up to home and even personal devices, keeping all connected on top of a new paradigm: everything runs now "on the cloud" (or in a more elegant term, be as much "Cloud Native" as possible) moving every possible infrastructure from on-premises devices to the public cloud boosting throughput with less expenses and surpassing the administrative responsibility to externals, leveraging at the same time staff capabilities to continuously improve its productivity.

However, this rapid advancement brings a stealthy malicious consequence behind, lurking silently behind the door: cyber attackers. More devices interconnected means more data exchanged publicly, with a wider attack surface vulnerable to threat actors that could exploit any possible vulnerability to gain access to the network, and even worse, to the data.

The description of how this threads are addressed and modeled is a another thesis topic itself, but in this project we are focusing on just on the network traffic analysis that could help during the final phase of an attack, or could be called the *post-mortem* phase, where the network administrator could analyze the traffic and determine the root cause of the attack, and even better, to prevent it from happening again.

This project is a initiative from the Data Network (DN) Laboratory from the University of Applied Sciences of Cologne, Germany; aims to provide tools to monitor and capture life traffic from a virtualized 5G environment, so a network administrator is provided with valid tools for a post-mortem analysis.

# Contents

# List of Tables

# List of Figures

# 1 Introduction

In this thesis, a complete study for evaluating virtual Test Access Points vTAP and monitoring alternatives for virtualized 5th Generation 5G environments is documents. This project is a initiative from the Data Network (DN) Laboratory from the University of Applied Sciences of Cologne, Germany; after multiple studies on the 5G core, focusing on the networking functioning and its continuous development over the years.

## 1.1 Objectives

The objectives of the current project are the following:

- Study the theoretical framework behind virtualization, containerization and orchestration of a 5G environment, as well as the monitoring and vTAP concepts needed for future sections.
- Evaluate the current state of the art of virtual Test Access Points vTAP for 5G environments.
- Build a working testbed of a 5G environment, virtualized and deployed in the private cloud from the DN Lab.
- Propose viable options for vTAP and for monitoring network traffic flowing across the 5G environment.
- Evaluate the functioning of the proposed solutions and give conclusions that could lead to further improvements and future work on the subject.

In the second chapter, the main core concepts will be introduced for the reader, so a successful understanding of the project can be achieved. The third chapter will present how the testbed was designed and implemented, and also some test of the right functioning of the target 5G functions. The fourth chapter encompass the implementation and test of each of the proposed solutions, and the a final chapter with conclusions and suggesting future work.

Cologne, June 2023

# 2 Fundamentals

In this chapter, a general overview of the concepts that will be used during the following chapters is introduced, so the common reader can get a formal introduction of the concepts used during every experimentation and evaluation phase.

## 2.1 5G network fundamentals

Mobile communication technologies have undergone remarkable transformations over the past few decades, transitioning from the initial analog systems of the first generation (1G) to the sophisticated digital and broadband capabilities of the fourth generation (4G). Each generational leap has been characterized by significant improvements in speed, capacity, and user experience, culminating in the current development and deployment of the fifth generation (5G) networks.

The first generation (1G) introduced analog voice communication, enabling mobile telephony for the first time. The second generation (2G) marked the shift to digital communication, improving voice quality and enabling text messaging services. The third generation (3G) brought mobile internet access, facilitating data services such as web browsing and email on mobile devices. The fourth generation (4G) further enhanced data speeds and capacity, enabling high-definition video streaming, video conferencing, and a wide range of mobile applications.

5G is poised to revolutionize the way we connect, communicate, and interact with our environment, offering unprecedented opportunities for innovation across various sectors, including healthcare, transportation, entertainment, and industrial automation. The evolution of mobile networks reflects a continuous effort to meet increasing user demands and technological advancements.

### 2.1.1 Core Concepts of 5G Technology

At its core, 5G technology is designed to meet the growing demands for high-speed data transfer, ultra-reliable low latency communication (URLLC), and massive machine-type communications (mMTC). These three primary use cases are the pillars upon

which 5G is built, addressing the needs of diverse applications ranging from enhanced mobile broadband (eMBB) to the Internet of Things (IoT) and critical infrastructure communications.

- **Enhanced Mobile Broadband (eMBB):** focuses on providing faster data speeds and more reliable internet connections. This aspect of 5G aims to support applications such as high-definition video streaming, virtual reality (VR), and augmented reality (AR), offering users seamless and immersive experiences. eMBB is essential for consumer applications requiring high bandwidth and low latency, transforming how we consume media and interact with digital content.

  The proliferation of high-definition and 4K video content, along with the emergence of VR and AR technologies, necessitates a robust and high-speed network infrastructure. 5G's eMBB capabilities ensure that users can experience uninterrupted streaming, real-time gaming, and immersive virtual environments without lag or buffering.

- **Ultra-Reliable Low Latency Communication (URLLC):** Ultra-Reliable Low Latency Communication (URLLC) is crucial for applications that require real-time data transmission with minimal delay. This includes autonomous vehicles, industrial automation, and remote medical procedures, where reliability and low latency are critical for safety and performance.

  For instance, in autonomous driving, URLLC ensures that vehicles can communicate with each other and with traffic infrastructure instantaneously, preventing accidents and optimizing traffic flow. In industrial automation, URLLC supports the coordination and control of robots and machinery, enhancing productivity and safety in manufacturing environments. Remote medical procedures, such as telesurgery, rely on URLLC to transmit precise control commands and high-definition video feeds, enabling surgeons to perform operations from a distance with high accuracy and minimal delay.

- **Massive Machine-Type Communications (mMTC):** Massive Machine-Type Communications (mMTC) addresses the need for a network capable of supporting a vast number of connected devices, often with low data rates. mMTC is essential for the proliferation of IoT devices, smart cities, and connected agriculture, where billions of sensors and devices need to communicate efficiently.

  In smart cities, mMTC enables the deployment of sensors for monitoring air quality, managing waste, and optimizing energy consumption. Connected agriculture benefits from mMTC by using sensors to monitor soil conditions, crop health, and equipment status, allowing farmers to make data-driven decisions that optimize irrigation, fertilization, and pest control. The ability to connect

a large number of devices simultaneously without network congestion is a key advantage of mMTC in 5G.

### 2.1.2 5G Network Concepts

To achieve the ambitious goals of 5G, the network architecture incorporates several advanced concepts and technologies that transform the way mobile networks are designed, deployed, and managed. These innovations enable 5G to deliver unprecedented performance, flexibility, and efficiency. Below, we explore these key network concepts in greater detail.

#### Network Slicing

Network slicing is a fundamental feature of 5G that allows a single physical network to be divided into multiple virtual networks, or "slices," each tailored to specific applications or user requirements. This concept is akin to creating multiple independent, end-to-end networks that share the same physical infrastructure but operate in isolation from one another.

Each network slice can be optimized for different performance characteristics, such as latency, bandwidth, reliability, and security. For instance:

- A slice dedicated to autonomous vehicles can prioritize ultra-low latency and high reliability to ensure safe and responsive communication.
- A slice for enhanced mobile broadband (eMBB) can focus on providing high data rates and large capacity for streaming high-definition video and other data-intensive applications.
- A slice for massive machine-type communications (mMTC) can support a vast number of IoT devices with low power consumption and efficient data handling.

Network slicing is made possible through advanced technologies such as Software-Defined Networking (SDN) and Network Function Virtualization (NFV), which allow for dynamic and flexible network management. SDN decouples the control plane from the data plane, enabling centralized and programmable network control. NFV virtualizes network functions, running them on general-purpose hardware rather than specialized equipment, which facilitates rapid deployment and scaling of network services.

### Massive MIMO (Multiple Input Multiple Output)

Massive MIMO is a technology that significantly enhances the capacity and efficiency of 5G networks by using a large number of antennas at both the transmitter and receiver. Unlike traditional MIMO systems that use a few antennas, massive MIMO systems can employ dozens or even hundreds of antennas.

The key advantages of massive MIMO include:

- **Increased Capacity:** By transmitting multiple data streams simultaneously, massive MIMO can support a higher number of users and devices in a given area, making it ideal for densely populated environments such as urban centers and stadiums.
- **Improved Spectral Efficiency:** Improved Spectral Efficiency: Massive MIMO improves the use of available spectrum, allowing for more data to be transmitted over the same frequency bands. This leads to better utilization of the limited radio spectrum and enhances overall network performance.
- **Enhanced Signal Quality:** The use of multiple antennas enables advanced signal processing techniques, such as spatial multiplexing and beamforming, which can improve signal strength and reduce interference. Beamforming, in particular, focuses the wireless signal towards specific users or devices, enhancing coverage and reliability.

Massive MIMO is a critical component of 5G networks, enabling them to meet the high capacity and performance demands of modern mobile communication.

### Edge Computing

Edge computing is a paradigm that brings data processing and storage closer to the location where it is needed, rather than relying on centralized data centers. This is particularly important for applications requiring real-time data processing, such as autonomous driving, industrial automation, and augmented reality.

The benefits of edge computing in 5G networks include:

- **Reduced Latency**: By processing data at the network edge, closer to the end user, edge computing minimizes the time it takes for data to travel back and forth between the user and the central data center. This is crucial for latency-sensitive applications that require immediate responses.

- **Enhanced Privacy and Security:** Processing data locally can mitigate the risks associated with transmitting sensitive information over long distances. Edge computing can also enable localized security measures, providing an additional layer of protection.
- **Improved Bandwidth Efficiency:** Offloading data processing tasks to edge nodes reduces the amount of data that needs to be transmitted over the core network, freeing up bandwidth for other applications and improving overall network efficiency.

Edge computing integrates with 5G networks through the deployment of edge nodes or servers at strategic locations, such as base stations or regional data centers. These edge nodes handle data processing tasks and provide services to users within their vicinity, ensuring low-latency and high-performance experiences.

### 2.1.3 Software-Defined Networking (SDN) and Network Function Virtualization (NFV)

SDN and NFV are key enablers of the flexibility and efficiency of 5G networks, allowing for more dynamic and programmable network management.

#### Software-Defined Networking (SDN)

- **Centralized Control:** SDN separates the control plane from the data plane, enabling centralized control and management of the network. This allows network operators to dynamically adjust network configurations and policies in response to changing demands and conditions.
- **Programmability:** SDN enables network programmability, allowing operators to automate network management tasks, deploy new services quickly, and optimize resource allocation. This flexibility is essential for supporting diverse 5G use cases and applications.

#### Network Function Virtualization (NFV)

- **Virtualized Network Functions:** NFV replaces traditional network appliances, such as routers and firewalls, with software-based functions that run on standard servers. This reduces the reliance on specialized hardware and enables more efficient use of resources.

- **Scalability and Agility:** NFV allows network functions to be deployed, scaled, and managed dynamically, providing the agility needed to respond to varying network conditions and user demands. Virtualized network functions can be instantiated and terminated as needed, ensuring optimal performance and cost-efficiency.

Together, SDN and NFV transform 5G networks into flexible, programmable, and scalable infrastructures capable of meeting the diverse and evolving needs of modern communication.

## 2.2 The Cloud Native Approach

The rise of cloud technologies has fundamentally altered our approach to hardware, system management, and physical networking. Virtual machines have taken over from physical servers, storage services have replaced discussions of hard drives, and automation tools have become more prominent. This shift marked an early stage in rethinking cloud technology. As we better understood the strengths and limitations of this new model, it also began to reshape how we design applications and services.

Developers and operators started to rethink the strategy of creating large, monolithic applications that run on powerful hardware. They saw the challenges in maintaining data integrity while sharing data across various applications. Issues like distributed locking, storage, and caching moved from academic discussion to mainstream challenges. Software was increasingly broken into smaller, independent executables. As Brendan Burns, the founder of Kubernetes, often states, "distributed computing has transitioned from a specialized subject to a fundamental topic in computer science."

The concept of "cloud native" reflects this shift in perception towards an architecture that leverages the cloud's potential and limitations. Designing systems with cloud capabilities and constraints in mind means creating cloud native systems.

In this section, some key components of the cloud native approach will be introduced, as they are the basis for the following chapters.

### 2.2.1 Containers and Microservices

At the core of cloud native computing is the belief that it's better to use multiple small, standalone services rather than a single large service that manages everything. Rather than creating a large application that handles everything from generating the user interface to processing tasks and managing databases and caches, the cloud native

philosophy advocates for developing a collection of specialized services. Each service performs a specific function and they work together to fulfill a broader purpose. For example, in such a setup, one service might exclusively manage a relational database. Other services needing access to this data would interact with it via a representational state transfer (REST) API, typically using JavaScript Object Notation (JSON) over HTTP to fetch and update information.

This method allows developers to conceal complex underlying implementations and focus on providing functionalities that align directly with the application's business goals.

### Microservices

In traditional setups, an application might have been a single executable handling all tasks, but in the cloud native landscape, applications are distributed across multiple separate programs. Each program is responsible for one or a few specific tasks, yet collectively, they constitute a single cohesive application.

To illustrate, consider an ecommerce website (as shown in the figure below). Various tasks such as managing a product catalog, handling user accounts and shopping carts, processing payments, and providing a customer interface could each be handled by separate components. In the past, these tasks might have been integrated into a single program running on robust hardware.

However, with a cloud native approach, this ecommerce system would be segmented into distinct services: one for payment processing, another for the product catalog, another for administration, and so forth. These services communicate over the network using clearly defined REST APIs. As an example, see the picture [15] below for a better understanding.

In its most extreme form, this approach breaks down an application into its smallest functional units, each represented by a separate program. This is known as microservice architecture. A microservice is designed to manage only a narrow aspect of the application's overall functionality, standing in stark contrast to the monolithic model.

The microservice approach has significantly shaped the development of cloud native computing, particularly evident in the rise of container technology.

### Containers

A common comparison in the tech world is between containers and virtual machines. While a virtual machine hosts a full operating system in an isolated environment on a

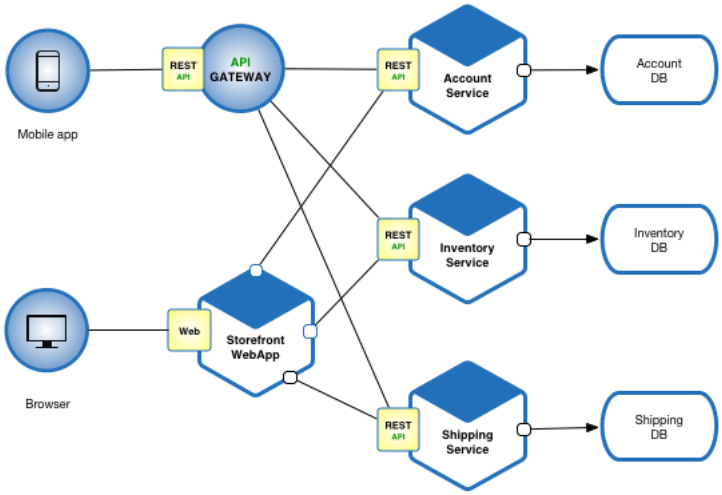



Figure 2.1: Microservices Architecture [15]

host machine, a container shares the host's operating system kernel but maintains its own filesystem.

Another way to think about containers—a perspective that may be more relevant to our discussion—is as a method for encapsulating a program's runtime environment. This packaging ensures that all dependencies are met when the program is transferred from one host to another.

This conceptualization introduces some non-technical limitations on how containers are ideally used. Although it's technically possible to run multiple programs [1] within a single container, containers—particularly as designed by Docker—are intended to house just one main program. [3]

Containers are ideally suited for microservices architecture. Each microservice, small and self-contained, can be packaged into a container along with all its dependencies, allowing easy transport from one host to another. Containers eliminate the need for hosts to maintain all the necessary tools to run the programs they contain. For instance, if a program requires Python 3, the host doesn't need to install Python or its libraries; everything is contained within the container.

[1] In this context, when we mention "programs," we're referring to a concept more abstract than just a binary file. For instance, a Docker container typically contains several executables that support the main application, but these are secondary to the container's primary function. A web server might include additional utilities for startup or other low-level operations, yet the main web server remains the focus of the container.

This setup allows hosts to run multiple containers with differing requirements without needing to manage these dependencies. For example, a Python 2 container can run alongside a Python 3 container without any conflict or additional setup by administrators.

All this changes led to a significant shift in the roles of administrators, operators, and site reliability engineers (SREs). They are no longer burdened with managing individual program dependencies. Instead, they can focus on allocating resources like network, storage, and CPU more effectively for these containerized applications.

While isolation is a key feature of containers, often we need to expose certain aspects of a container to the outside world, such as network access, storage, or specific configurations. The container runtime handles these interactions, allowing a container to integrate into a larger environment, which might include other containers on the same host or services across the network.

**Container Images and Registries** Container technology extends into a sophisticated realm of its own. A container is essentially a program packaged with its dependencies and environment into a portable format known as a container image. These images are not single binaries but are composed of layered segments, each identifiable by a unique identifier. When an image needs to be transferred, only the missing layers need to be fetched, streamlining the process significantly.

A pivotal component in this system is the image registry, a specialized storage solution for housing container images. It enables hosts to upload and download container images, managing the individual layers and ensuring that hosts only download the layers they lack.

Registries identify images using three key pieces of information:

- **Name:** This can vary from simple (e.g., nginx) to complex personalized setups including a dedicated registry URL (e.g., example.com/servers/nginx), depending on the registry.
- **Tag:** Typically denotes the software version (e.g., v1.2.3), but can also be a generic label like 'latest' or 'stable' to indicate the most current or production-ready version.
- **Digest:** To ensure accuracy in version control, since tags can change, registries also use a digest—a hash sum of the image's layers—to pinpoint a specific version.

> In summary, while container technology involves complex elements like images and registries, these concepts set the stage for understanding more advanced topics like Kubernetes, which we'll explore in the context of schedulers.

However, containers and microservices brought a new challenge to system administrators and IT technicians in general. How to manage and orchestrate all these containers and microservices in a efficient and scalable way? This is where Kubernetes comes in, and its fundamentals will be explained in the next section.

## 2.3 Kubernetes Fundamentals

Kubernetes, often abbreviated as K8s, is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. Originally developed by Google, Kubernetes has become the de facto standard for container orchestration, supported by a robust community and maintained by the Cloud Native Computing Foundation (CNCF). This section delves into the fundamental concepts of Kubernetes, providing a comprehensive understanding of its architecture, components, and operational principles.

### 2.3.1 Background and Evolution

The shift towards containerization revolutionized software development and deployment by enabling applications to run consistently across different environments. Containers package an application and its dependencies into a single unit, ensuring that it behaves the same, regardless of where it is deployed. This consistency addresses the common "it works on my machine" problem, but managing containers at scale introduces significant complexity. This complexity necessitated the development of container orchestration platforms to handle tasks such as deployment, scaling, networking, and storage.

Kubernetes emerged from Google's extensive experience with container management through its internal Borg system. Launched in 2014, Kubernetes has rapidly evolved to offer a rich set of features addressing the complexities of container orchestration. Its design principles emphasize scalability, resilience, and ease of use, making it the preferred choice for modern cloud-native applications. Over the years, Kubernetes has integrated numerous advanced features and maintained strong community support, ensuring its relevance and adaptability in a rapidly changing technological landscape.

### 2.3.2 Kubernetes Architecture

Kubernetes architecture is meticulously designed to ensure high availability, scalability, and maintainability of containerized applications. It comprises several key components, each playing a crucial role in the orchestration process, ensuring the smooth operation and management of the cluster. The common architecture is that a kubernetes cluster is composed by a control plane and multiple worker nodes. These concepts and its main components will be described in the next sections and depicted in the following figure

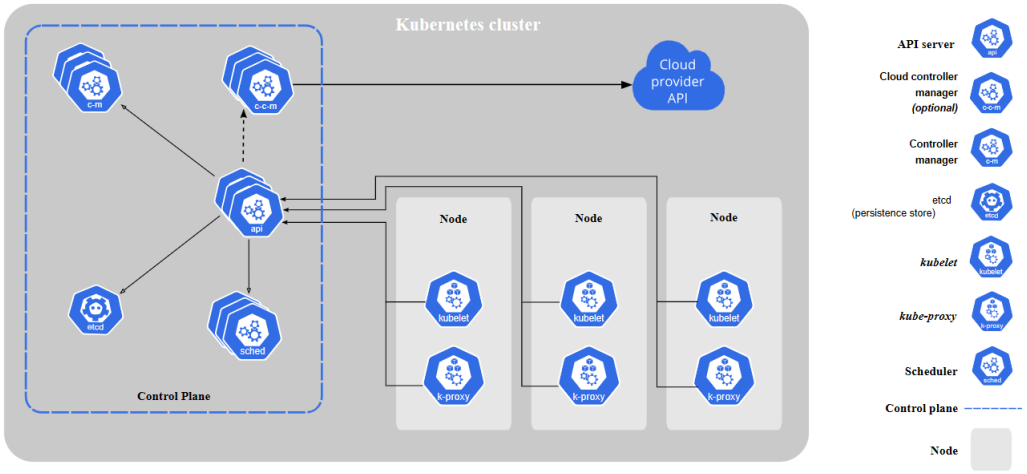



Figure 2.2: Kubernetes Architecture [12]

#### Control Plane

The control plane manages the overall state and operations of the Kubernetes cluster. These components include:

**API Server** The API server acts as the central management point for the cluster, exposing the Kubernetes API. It processes RESTful requests, validates them, and updates the cluster's state in the etcd store. The API server is designed to handle large volumes of requests, ensuring responsive and efficient management. It serves as the gateway through which all administrative tasks are performed, from creating deployments to scaling applications.

**etcd** etcd is a distributed key-value store that serves as the cluster's backing store. It stores all cluster data, including configuration details, state information, and metadata. etcd is designed for high availability and consistency, ensuring that the cluster state is reliably maintained. Its distributed nature ensures data redundancy and fault tolerance, critical for the robust operation of the cluster.

**Controller Manager** The controller manager runs various controller processes that regulate the state of the cluster. Controllers continuously monitor the cluster state and make adjustments to ensure the desired state is maintained. For example, the replication controller ensures the specified number of pod replicas are running at all times. Other controllers manage tasks like node operations, endpoint tracking, and namespace management, each contributing to the self-healing and adaptive nature of Kubernetes.

**Scheduler** The scheduler is responsible for assigning newly created pods to nodes within the cluster. It evaluates the resource requirements of the pods against the available resources on the nodes, making decisions based on various factors such as resource utilization, affinity rules, and constraints. The scheduler's role is crucial for optimizing resource use and ensuring that applications meet their performance and resource requirements.

### Node Components

Nodes are the worker machines in a Kubernetes cluster, where the containerized applications run. Each node includes several critical components:

**Kubelet** The kubelet is an agent that runs on each node, responsible for ensuring that containers are running in pods. It communicates with the API server to receive instructions and manage the lifecycle of pods on its node. The kubelet ensures that the correct containers are running and that they adhere to the defined specifications, managing tasks such as health checks, logging, and container monitoring.

**Kube-proxy** Kube-proxy is a network proxy that runs on each node, managing network communication for the pods. It maintains network rules on the nodes, enabling communication between different pods within the cluster and managing external access. Kube-proxy handles service discovery and load balancing, ensuring that traffic is routed efficiently and correctly within the cluster.

**Container Runtime** The container runtime is the software responsible for running containers. Kubernetes supports various container runtimes, including Docker, containerd, and CRI-O. The container runtime pulls container images, starts, and stops containers as instructed by the kubelet. This modularity allows Kubernetes to support a range of container technologies, enhancing its flexibility and compatibility.

### Kubernetes Objects and Resources

Kubernetes manages applications through a set of API objects, which represent the desired state of various cluster resources. Key Kubernetes objects include:

**Pod** A pod is the smallest and simplest Kubernetes object. It represents a single instance of a running process in the cluster. Pods can contain one or more containers, which share the same network namespace and storage volumes. Pods are ephemeral, meaning they can be created, destroyed, and recreated dynamically. This ephemeral nature supports flexible scaling and rapid recovery from failures.

**Service** A service is an abstraction that defines a logical set of pods and a policy for accessing them. Services provide stable endpoints for applications, decoupling the client from the dynamic nature of pod IP addresses. They support load balancing, ensuring even distribution of traffic across pods, and facilitate service discovery within the cluster.

**ReplicaSet** A ReplicaSet ensures a specified number of pod replicas are running at any given time. It provides self-healing capabilities, automatically creating new pods to replace failed ones. ReplicaSets are often used indirectly through higher-level abstractions like Deployments. They ensure that applications maintain their desired state, even in the face of node failures or other disruptions.

**Deployment** A deployment is a higher-level abstraction that manages the lifecycle of ReplicaSets and pods. It provides declarative updates, allowing users to define the desired state of an application and handles rolling updates, rollbacks, and scaling. Deployments make it easier to manage complex application updates and ensure zero-downtime deployments, enhancing the reliability and agility of application management.

**StatefulSet** A StatefulSet is similar to a Deployment but is designed for stateful applications. It ensures that pod replicas have unique, stable identities and persistent storage, making it suitable for applications like databases and distributed systems. StatefulSets provide guarantees about the ordering and uniqueness of pods, essential for applications that require stable network identifiers and storage persistence.

**ConfigMap and Secrets** ConfigMaps and Secrets are used to manage configuration data and sensitive information, respectively. ConfigMaps store non-confidential configuration data as key-value pairs, while Secrets store sensitive data, such as passwords and API keys, in a secure manner. These objects allow for dynamic configuration of applications without the need to rebuild container images.

### Operational Concepts

Effective operation of Kubernetes requires understanding several key concepts that ensure the smooth functioning and management of the cluster.

**Namespaces** Namespaces provide a mechanism for isolating groups of resources within a single Kubernetes cluster. They are useful for dividing cluster resources among multiple users or teams, enabling resource allocation and access control. Namespaces can also help prevent naming conflicts by creating distinct environments within the cluster. This isolation supports multi-tenant environments and simplifies resource management across diverse teams and projects.

**Labels and Selectors** Labels are key-value pairs attached to Kubernetes objects, enabling users to organize and select subsets of objects based on specific criteria. Selectors allow users to query and manage objects based on their labels. Labels and selectors are fundamental for grouping and managing resources, facilitating operations like scaling, updating, and monitoring. They provide a flexible way to organize resources according to any dimension, such as environment, version, or application component.

**Resource Quotas and Limits** Resource quotas and limits ensure fair resource allocation and prevent resource exhaustion within a cluster. Quotas set constraints on the number of resources that can be consumed by a namespace, while limits define the maximum resources that individual pods or containers can use. These mechanisms help maintain cluster stability and ensure that critical applications have the necessary

resources to operate effectively. Resource quotas and limits are essential for preventing resource contention and ensuring predictable performance across different workloads.

**Role-Based Access Control (RBAC)** RBAC is a security mechanism that regulates access to Kubernetes resources based on user roles and permissions. It provides fine-grained control over who can perform specific actions within the cluster. RBAC policies define roles, which specify sets of permissions, and role bindings, which assign roles to users or groups. Implementing RBAC helps secure the cluster and enforce organizational policies. RBAC is crucial for maintaining a secure and compliant Kubernetes environment, especially in multi-tenant and large-scale deployments.

### Advanced Kubernetes Features

Kubernetes offers several advanced features that enhance its capabilities and support complex application requirements.

**Horizontal Pod Autoscaling** Horizontal Pod Autoscaling (HPA) automatically adjusts the number of pod replicas based on observed metrics, such as CPU utilization or custom metrics. HPA ensures that applications can scale dynamically in response to changing workloads, maintaining performance and resource efficiency. This feature is essential for applications with variable workloads, ensuring they can handle peak demand without overprovisioning resources during low-usage periods.

**Persistent Storage** Kubernetes provides robust support for persistent storage, enabling stateful applications to retain data across pod restarts. Persistent Volume (PV) and Persistent Volume Claim (PVC) abstractions decouple storage from individual pods, allowing for flexible storage management. Kubernetes supports various storage backends, including cloud storage services, network file systems, and local disks. This flexibility ensures that applications can meet their data persistence requirements regardless of the underlying infrastructure.

**Ingress** Ingress is a collection of rules that govern how external traffic is routed to services within the Kubernetes cluster. It provides a way to expose HTTP and HTTPS routes, enabling fine-grained control over traffic routing, load balancing, and SSL termination. Ingress controllers, such as NGINX or Traefik, manage ingress resources and ensure efficient traffic management. Ingress enhances the accessibility and performance of applications by providing a unified approach to managing external access.

### 2.3.3 Custom Resource Definitions

Custom Resource Definitions (CRD) are a powerful feature of Kubernetes that allows users to extend the Kubernetes API to support custom resources. This capability provides a flexible mechanism to manage application-specific objects without modifying the core Kubernetes code. By leveraging CRDs, organizations can tailor Kubernetes to meet their unique requirements, creating custom workflows and integrations that enhance the platform's utility and adaptability.

At its core, a Custom Resource Definition is a specification that defines a new resource type in the Kubernetes API. Once a CRD is registered, users can create and manage instances of this new resource type, known as Custom Resources (CRs), in the same way they manage built-in Kubernetes resources like Pods, Services, and Deployments. This extensibility is crucial for supporting complex, domain-specific application needs that are not covered by the default Kubernetes resource types.

#### Benefits of Using CRDs

By extending the Kubernetes' API, developers team can easily adapt and fit their own development and application into the kubernetes core, which has enabled the community to deliver a great set of custom resources for different applications and use cases, some of the best known are:

- **Prometheus Operator:** The Prometheus Operator is a popular CRD that simplifies the deployment and management of Prometheus monitoring instances in Kubernetes. It provides a declarative way to define monitoring configurations, alerting rules, and service discovery, streamlining the monitoring setup process.
- **Cert-Manager:** Cert-Manager is a CRD that automates the management of TLS certificates in Kubernetes. It enables users to request, issue, renew, and revoke certificates from various certificate authorities, ensuring secure communication between applications and services.
- **Istio:** Istio is a service mesh platform that leverages CRDs to enhance traffic management, security, and observability in Kubernetes clusters. It enables users to define traffic routing rules, implement security policies, and collect telemetry data using custom resources.
- **Argo CD:** Argo CD is a CRD that automates the continuous delivery of applications in Kubernetes. It provides a declarative way to define application manifests, synchronize them with a Git repository, and manage application deployments across multiple clusters.

The list can be further long, but the main idea is that CRDs are a powerful tool for extending Kubernetes' capabilities and adapting the platform to meet diverse application requirements. By defining custom resources, users can create domain-specific abstractions, automate complex workflows, and integrate third-party services seamlessly into their Kubernetes environments.

At its core, a Custom Resource Definition is a specification that defines a new resource type in the Kubernetes API. Once a CRD is registered, users can create and manage instances of this new resource type, known as Custom Resources (CRs), in the same way they manage built-in Kubernetes resources like Pods, Services, and Deployments. This extensibility is crucial for supporting complex, domain-specific application needs that are not covered by the default Kubernetes resource types.

### Benefits of Using CRDs

CRDs offer several significant benefits:

- **Extensibility:** CRDs enable users to add new resource types to Kubernetes, expanding its functionality to support custom use cases. This extensibility allows Kubernetes to manage a broader range of applications and services.
- **Declarative API:** CRDs leverage the same declarative API model as built-in resources. This consistency simplifies the management of custom resources and integrates seamlessly with existing Kubernetes tools and practices.
- **Automation with Custom Controllers:** CRDs can be paired with custom controllers to automate the lifecycle management of custom resources. Controllers watch for changes to custom resources and reconcile their actual state with the desired state, ensuring that applications behave as expected.
- **Reusability and Sharing:** Once defined, CRDs can be reused across multiple clusters and shared within the community. This reusability promotes standardization and collaboration, enabling organizations to build on each other's work.

The process of creating and managing would be briefly described in future sections, as this chapter is focused on the concepts and fundamentals of the technologies used in the following chapters. Following another important concept for the purpose of this project will be introduced, which is custom controllers.

### 2.3.4 Custom Controllers

Custom controllers are an essential component when working with CRDs, as they provide the automation needed to manage custom resources effectively. A custom controller is a Kubernetes component that continuously watches for changes to custom resources and takes appropriate actions to reconcile their actual state with the desired state. This reconciliation process is at the heart of Kubernetes' declarative model, ensuring that the cluster operates as intended. Therefore, a custom controller is the entity responsible for handling and preserving the *control loop* for a new custom resource. The image below tries to interactively depict how the control loop works in kubernetes:

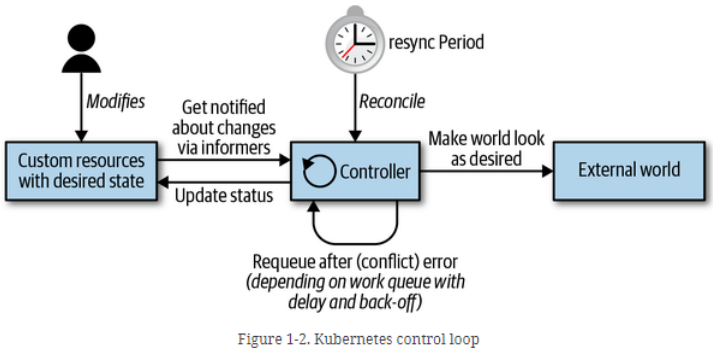



Figure 2.3: Kubernetes Control Loop [7]

The controller must watch for changes in the custom resource, evaluate the current state, and take actions to ensure that the resource remains in the desired state. This process is repeated continuously to maintain the resource's integrity and respond to any changes or events that affect its state. Custom controllers are typically implemented as Kubernetes controllers, which are control loops that watch for changes to resources and take actions to reconcile their state. The controller pattern is a fundamental concept in Kubernetes, enabling users to automate resource management and enforce desired configurations.

To effectively manage custom resources, custom controllers follow a common pattern:

- **Watch Custom Resources:** The watching mechanism is a core part of the custom controller's functionality. Informers are used to monitor the state of resources continuously. They watch for events such as resource creation, updates, and deletions. When an event occurs, the informer notifies the controller, which then processes the event. This continuous monitoring ensures that the controller is always aware of the state of the custom resources it manages.

Informers are efficient because they use mechanisms like caches to reduce the load on the Kubernetes API server. They store a local copy of the resource state, which they keep up-to-date by listening for changes. This reduces the need for frequent API calls and improves the performance of the controller.

- **Evaluate:** Once the controller is notified of a change, it evaluates the current state of the custom resource. This involves comparing the observed state (the actual state of the resource in the cluster) with the desired state (the state specified in the resource's spec). The controller's logic determines what actions, if any, need to be taken to reconcile these two states.

  This evaluation process often involves validating the resource against a set of rules or policies. For example, the controller might check that the resource's configuration is valid, that required fields are present, and that the resource conforms to any constraints defined by the application or system.

- **Take Action:** Based on the evaluation, the controller takes actions to reconcile the observed state with the desired state. These actions can include creating, updating, or deleting resources, as well as performing specific operations required by the application. The actions are designed to ensure that the resource reaches and maintains the desired state.

  For instance, if a custom resource represents a database cluster and the desired state specifies a certain number of replicas, the controller might create additional replicas if there are too few, or delete replicas if there are too many. The controller might also handle tasks such as updating configurations, performing backups, or managing failover processes.

  Custom controllers often use Kubernetes client libraries to perform these actions. These libraries provide a set of functions and methods for interacting with the Kubernetes API, making it easier to manage resources programmatically.

Another important concern while designing a custom controller is error handling. Custom controllers should be robust and resilient to failures, ensuring that they can recover from errors and continue operating effectively. This includes implementing mechanisms for logging, monitoring, and handling exceptions, as well as incorporating retry logic and backoff strategies to manage transient errors; but it is not directly related to Kubernetes concepts, as it should be treated as another consideration during software development process.

By introducing custom controllers, organizations can automate complex workflows, enforce policies, and integrate custom logic into their Kubernetes environments. Custom controllers are a powerful tool for extending Kubernetes' capabilities and adapting the platform to meet diverse application requirements. They enable users

to define custom resources, automate lifecycle management, and implement domain-specific functionality, enhancing the flexibility and utility of Kubernetes. In future sections, a whole process of creating a custom controller for the particular use case of setting up a virtual Traffic Access Point (vTAP) will be described and implemented.

## 2.4 Helm

Helm is usually defined as a package manager for Kubernetes, but it is much more than that. Helm is a tool that streamlines the installation and management of Kubernetes applications. It provides a templating engine to define Kubernetes manifests, enabling users to create reusable, shareable, and version-controlled application packages. Helm simplifies the deployment process by encapsulating application configurations and dependencies into a single package, known as a chart. This section will provide an overview of Helm's key features and components, highlighting its role in managing Kubernetes applications.

Similar to Linux package managers like APT or YUM, Helm simplifies the installation and management of Kubernetes applications by providing a standardized way to define, package, and deploy applications. Helm charts encapsulate all the necessary Kubernetes resources, configurations, and dependencies required to deploy an application, making it easy to share and distribute applications across different environments. Helm's templating engine allows users to define dynamic configurations, enabling the creation of reusable and customizable charts that can be easily deployed and managed.

### 2.4.1 Helm's Key Concepts

Helm introduces several key concepts that are essential to understanding its functionality and usage:

#### Helm Chart

In Helm's terms, a *Chart* is a package and it streamlines the way a Kubernetes application should be installed. A chart is a set of files and directories that adhere to the chart specification for describing the resources to be installed into Kubernetes. [3]

A Helm chart consists of the following components:

- **Chart.yaml:** The Chart.yaml file contains metadata about the chart, including the chart name, version, description, and other details. This file provides essential information about the chart and its contents.
- **Values.yaml:** The Values.yaml file defines default configuration values for the chart. These values can be overridden during installation, allowing users to customize the chart's behavior without modifying the chart itself. Values.yaml is a key component of Helm's templating engine, enabling dynamic configuration based on user input.
- **Templates:** The Templates directory contains Kubernetes manifest files that define the resources to be deployed. These manifest files can include deployments, services, config maps, and other Kubernetes resources. Helm's templating engine processes these files, allowing users to define dynamic configurations and reusable templates.
- **Other files:** Some other files include the `NOTES.txt` which is rendered and then displayed after a successful installation, and some other auxiliary files that can be used for testing purposes or streamlining some other processes.

#### Helm's Resources, Installations and Releases

Helm manages Kubernetes applications through the concepts of resources, installations, and releases:

**Resources** Resources are the Kubernetes objects that make up an application. It could consists on just a few pods that expose a service and have a basic configuration, or it could be a complex system with tens or even hundreds of resources coupled together across many kubernetes clusters. By using helm, this application is handled as a single *Helm installation*.

**Installations** It is a group of resources rolled out into a kubernetes cluster by a single `helm install` command. Therefore, a user could have multiple `installations` of the same application (i.e. a web server) across different namespaces or even different clusters. Each installation is identified by a unique *release* name, which allows users to manage and track the application's lifecycle independently.

**Releases** A helm release is an instance of a chart deployed into a Kubernetes cluster. It represents a specific deployment of an application with a unique release name. Releases are versioned and can be upgraded, rolled back, or deleted independently. Helm tracks the state of each release, allowing users to manage and monitor their applications effectively. A user could have, for example, the release `0.4.5` of a `Open5Gs` helm chart, and then upgrade it to a new release by running the `helm upgrade` command. Therefore, a user could easily roll out new versions of an application while waiting for the old version to be deleted.

## 2.5 Network Monitoring

Network monitoring is an essential discipline within IT management, playing a crucial role in ensuring the performance, security, and reliability of networked systems. By continuously observing, measuring, and analyzing network traffic and performance metrics, network monitoring allows administrators to detect anomalies, diagnose issues, and maintain optimal network operations. This proactive approach is vital for preemptively identifying and resolving problems, thereby minimizing downtime and maintaining the quality of service for users and applications.

Network monitoring underpins several critical functions within an organization's IT infrastructure. Primarily, it enables performance optimization by providing insights into key metrics such as bandwidth usage, latency, and packet loss. By monitoring these metrics, administrators can optimize network configurations to ensure efficient data flow and high performance.

Security is another vital aspect of network monitoring. Continuous observation helps in detecting suspicious activities, such as unusual traffic patterns or unauthorized access attempts, which may indicate security breaches or malicious activities. This enables quick response to potential threats, enhancing the security posture of the network.

### 2.5.1 Key Components of Network Monitoring

Network monitoring systems consist of several interconnected components. Monitoring tools, either software applications or hardware devices, are responsible for collecting and analyzing network data. These tools use protocols such as Simple Network Management Protocol (SNMP), NetFlow, and sFlow to gather data from network devices. Metrics and logs provide detailed records of network events and performance indicators, such as bandwidth usage, latency, jitter, and error rates.

Dashboards and alerting systems are integral to network monitoring, providing real-time data visualizations and notifications. These tools help administrators interpret network status at a glance and respond to significant events or threshold breaches promptly. Effective network monitoring also relies on established protocols and standards to ensure accurate and consistent data collection

In the context of this project, the network monitoring main concern is to provide useful data for cybersecurity and forensic analysts, which will fed from the vTAPs and other network monitoring tools to provide a comprehensive view of the network traffic and events and be able to successfully determined which assets and attacker could have either accessed, corrupted, encrypted; or in general, compromised, or attempted to compromise. Since the study is focused on 5G virtual core network in a Kubernetes environment, the main challenges of network monitoring in this context will be described in the following sections.

### 2.5.2 Challenges of Network Monitoring in Kubernetes

Kubernetes has revolutionized the way applications are deployed and managed, providing a highly flexible and scalable platform for container orchestration. However, with this flexibility comes a set of unique challenges, particularly in the realm of networking. These challenges arise due to the dynamic, ephemeral, and highly distributed nature of Kubernetes environments, necessitating robust and sophisticated networking solutions. This section delves into the primary challenges of Kubernetes networking, exploring the complexities and offering insights into potential strategies for mitigation.

#### Dynamic Nature of Kubernetes Environments

The dynamic nature of Kubernetes environments is one of the most significant challenges. In Kubernetes, applications are deployed as containers within pods, which can be scaled up or down based on demand. This scaling is often automated, driven by metrics such as CPU and memory usage. Additionally, pods can be rescheduled across nodes in the cluster to optimize resource utilization or maintain high availability.

This constant flux creates challenges in maintaining consistent network configurations and ensuring that network policies are correctly applied as pods come and go. Traditional static IP addressing and network configurations are not feasible in such an environment. Instead, Kubernetes uses a more dynamic approach, assigning IP addresses to pods at the time of their creation, which requires sophisticated networking solutions that can handle these changes seamlessly.

### Complex Microservices Architectures

Kubernetes is particularly well-suited for microservices architectures, where applications are broken down into smaller, independently deployable services. While this architecture offers numerous benefits, including improved scalability and maintainability, it also introduces significant networking challenges.

Microservices often communicate extensively over the network, leading to a substantial amount of east-west traffic (internal traffic within the data center). Managing this traffic efficiently is critical to maintaining application performance and reliability. Network latency, jitter, and packet loss can significantly impact the performance of microservices, especially those that require low-latency communications.

Moreover, the interdependencies between microservices can complicate network configurations and troubleshooting. Understanding and managing these dependencies requires comprehensive network monitoring and tracing capabilities to visualize service interactions and identify potential bottlenecks or failure points.

### Ephemeral Nature of Containers

The ephemeral nature of containers, which are designed to be transient and short-lived, presents additional networking challenges. Containers can be created and destroyed rapidly, often within seconds. This ephemerality is a key feature of containerized applications, enabling rapid scaling and efficient resource utilization.

However, it also means that network configurations must be highly dynamic and capable of adapting in real-time. Network policies, security rules, and monitoring configurations must be consistently applied to new containers as they are created. Ensuring that these configurations are automatically and accurately applied is crucial for maintaining network security and performance.

### Network Security Concerns

Network security is a paramount concern in Kubernetes environments. The dynamic and distributed nature of Kubernetes clusters makes them particularly susceptible to security threats, including unauthorized access, data breaches, and internal attacks.

Implementing robust network security measures is essential to protect sensitive data and ensure compliance with regulatory requirements. This includes enforcing network segmentation to isolate different workloads, applying network policies to control traffic flow, and using encryption to secure data in transit.

Kubernetes provides several built-in security features, such as Network Policies, which allow administrators to define rules for controlling traffic between pods. However, configuring these policies correctly can be complex, requiring a deep understanding of both Kubernetes networking and the specific security requirements of the applications being deployed.

### Network Performance Optimization

Optimizing network performance in Kubernetes environments is another significant challenge. The distributed nature of Kubernetes clusters, often spanning multiple nodes and even data centers, can lead to performance issues such as high latency and bandwidth constraints.

Effective network performance optimization involves several strategies, including:

- **Efficient Routing:** Ensuring that network traffic is routed efficiently within the cluster to minimize latency and avoid bottlenecks.
- **Load Balancing:** Distributing traffic evenly across available resources to prevent any single node or service from becoming a bottleneck.
- **Quality of Service (QoS):** Implementing QoS policies to prioritize critical traffic and ensure that high-priority applications receive the necessary resources.
- **Network Policies:** Defining and enforcing network policies to control traffic flow and prevent unauthorized access.

### Observability and Troubleshooting

Achieving comprehensive observability and effective troubleshooting in Kubernetes environments is essential for maintaining network health and performance. Given the dynamic and distributed nature of Kubernetes, traditional monitoring and troubleshooting tools may fall short.

Effective observability involves collecting and analyzing a wide range of metrics, logs, and traces from various components of the Kubernetes cluster, including nodes, pods, services, and network interfaces. This data provides insights into network performance, identifies potential issues, and helps diagnose problems.

Advanced observability tools and techniques, such as distributed tracing and service meshes, can provide deeper visibility into the interactions between microservices and the flow of network traffic. These tools are invaluable for identifying performance

bottlenecks, understanding service dependencies, and pinpointing the root cause of network issues.

### 2.5.3 Strategies for Effective Kubernetes Network Monitoring

As in every solution in the IT world, there is not a single solution that fits all the cases, and the same applies to network monitoring in Kubernetes environments. However, there are several strategies and best practices that can help organizations effectively monitor their Kubernetes networks and address the challenges outlined above. By combining these strategies with the right tools and technologies, administrators can gain comprehensive visibility into their network operations, optimize performance, enhance security, and troubleshoot issues effectively.

Some of the best strategies for effective Kubernetes network monitoring include:

#### Integration with Kubernetes-Native Tools

Integrating with Kubernetes-native tools is a foundational approach to effective network monitoring. Kubernetes provides built-in tools and APIs that facilitate network monitoring. These native tools offer significant advantages, including seamless integration with the Kubernetes ecosystem, reduced complexity, and improved visibility into cluster operations. Some of the most relevant Kubernetes-native tools for network monitoring include:

**Kubernetes Metrics Server:** The Kubernetes Metrics Server is a cluster-wide aggregator of resource usage data. It provides metrics like CPU and memory usage for nodes and pods. By deploying the Metrics Server, administrators gain access to real-time resource usage statistics, which are essential for monitoring the overall health of the cluster and for making informed decisions about scaling and resource allocation.

**kubectl Top:** kubectl top is a command-line utility that interfaces with the Metrics Server to provide resource usage statistics directly in the terminal. It allows administrators to quickly check the resource consumption of nodes and pods, making it easier to identify resource bottlenecks and high-utilization areas that might need attention.

**Kubernetes Dashboard:** The Kubernetes Dashboard is a web-based user interface that provides an overview of applications running in the cluster, as well as the overall health of the cluster. It displays metrics collected by the Metrics Server, giving administrators a visual representation of resource usage and performance. The Dashboard also allows for direct interaction with Kubernetes objects, such as deploying applications, monitoring workloads, and managing cluster resources.

Another interesting approach when the native tools are not enough is to use third-party tools, which will be described in the following sections.

### Leveraging Service Meshes

A service mesh is a software layer that handles all communication between services in applications. Service meshes provide advanced networking features such as load balancing, service discovery, encryption, and observability. They are particularly well-suited for microservices architectures, where applications are composed of multiple services that communicate over the network.

Service meshes are dedicated infrastructure layers [16] that manage service-to-service communication within a microservices architecture. They provide advanced networking features such as load balancing, service discovery, encryption, and observability. Istio and Linkerd are two of the most popular service meshes used in Kubernetes environments.

**Istio:** Istio is a service mesh—a modernized service networking layer that provides a transparent and language-independent way to flexibly and easily automate application network functions. It is a popular solution for managing the different microservices that make up a cloud-native application. It includes traffic management, security, and observability components. Istio's observability features are particularly beneficial for network monitoring. It collects telemetry data from the Envoy sidecars injected into each service, providing metrics, logs, and traces that give detailed insights into the interactions between services. Administrators can use this data to monitor service performance, detect anomalies, and troubleshoot issues.

**Linkerd:** Linkerd is a lightweight service mesh designed for simplicity and performance. It provides many of the same features as Istio, including observability, security, and traffic management. Linkerd's observability capabilities include automatic collection of service metrics, such as request rates, success rates, and latencies. These metrics are aggregated and exposed via a Prometheus-compatible endpoint, making it easy to integrate with existing monitoring solutions.

**Prometheus and Grafana** Prometheus and Grafana are widely used together to provide a comprehensive monitoring and visualization solution for Kubernetes environments.

- **Prometheus:** Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability. It collects metrics from various sources, including Kubernetes nodes, pods, and services, and stores them in a time-series database. Prometheus supports powerful querying capabilities through its PromQL language, enabling detailed analysis of collected metrics. It also includes an alerting mechanism that allows administrators to define alerting rules based on specific conditions, ensuring that they are notified promptly of any critical events.
- **Grafana:** Grafana is a powerful visualization tool that integrates seamlessly with Prometheus. It allows administrators to create interactive and customizable dashboards to visualize the metrics collected by Prometheus. Grafana supports a wide range of graph types and visualization options, making it easy to create detailed and informative dashboards. Additionally, Grafana's alerting system can trigger notifications based on predefined conditions, ensuring that administrators are aware of any significant changes in the network's state.

**Logging:** Effective logging involves collecting and storing logs from various sources, including nodes, pods, and applications. The ELK stack (Elasticsearch, Logstash, and Kibana) is a popular logging solution for Kubernetes. Fluentd is often used to collect logs and forward them to Elasticsearch, where they can be indexed and searched. Kibana provides powerful visualization and analysis capabilities, enabling administrators to explore log data and identify issues.

**Tracing:** Distributed tracing tools, such as `Jaeger` and `Zipkin`, provide detailed visibility into the flow of requests across microservices. They collect trace data from applications and visualize it in a way that shows the end-to-end journey of a request. This helps identify latency issues, understand service dependencies, and pinpoint the root cause of performance problems. Integrating tracing with logging and metrics provides a comprehensive observability solution, allowing administrators to correlate data from different sources and gain deeper insights into network performance.

A final and brief introduction to virtualized 5G options will be provided in the following section.

### 2.5.4 Test Access Points (TAPs)

Test Access Points (TAPs) are network devices that provide a way to monitor and capture network traffic for analysis and troubleshooting purposes. TAPs are commonly used in network monitoring and security operations to gain visibility into network traffic, detect anomalies, and investigate security incidents. By passively monitoring traffic, TAPs enable administrators to analyze network behavior, identify performance issues, and respond to security threats effectively.

TAPs operate by copying network packets from the network link and forwarding them to monitoring tools, such as intrusion detection systems (IDS), packet analyzers, and network performance monitoring tools. This copy-and-forward mechanism ensures that monitoring tools receive an exact replica of the network traffic, without affecting the original data flow. TAPs are typically deployed at strategic points in the network, such as between switches, routers, or firewalls, to capture traffic from specific segments or devices.

Some books state that the only valid form of TAP is an inline-placed devices [6] as shown in the figure below, which are devices that are placed in the network path and can actively block or modify traffic. However, for the purpose of this project, the term TAP will be used to refer to any device that can capture and forward network traffic for monitoring and analysis. But the industry has changed and with the shift to cloud native applications, such a "device" could have multiple forms, as follows:

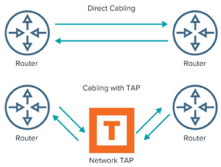



Figure 2.4: Direct cabling vs. TAP cabling [8]

#### Passive TAPs

It is only applicable for optical networks, as it is a passive TAP requires no power of its own and does not actively interact with other components of the network. It uses an optical splitter to create a copy of the signal and is sometimes referred to as a “photonic” TAP. Most passive TAPs have no moving parts, are highly reliable and do not require configuration.

The idea is that the device is placed inside the optical path of the network, and it acts as a passive splitter, meaning, it physically diverts a portion of the light from its original source, so the actual physical data can be studied. Since it is a passive device, it does not require power, and it does not introduce any latency or packet loss to the network, however, it reduces the original signal power, so plenty of physical layer conditions must be considered when deciding the use of this kind of TAP; like the following:

- Transmit power (the starting light signal).
- Receiver sensitivity (residual light seen at the other end).
- Light loss within the cable plant (prior to TAP insertion).
- Impact of the TAP (the actual TAP signal loss).

### Active TAPs

Active TAPs are devices that require power and actively interact with the network. They are typically placed inline with the network link and can selectively copy and forward traffic to monitoring tools. Active TAPs offer more flexibility and control over the traffic being monitored, allowing administrators to filter, aggregate, and manipulate packets before forwarding them to monitoring tools.

In the figure below, the basic functioning of an active TAP is shown, where the device is placed in the network path and copies the traffic to the monitoring tool, while the original traffic continues its path.

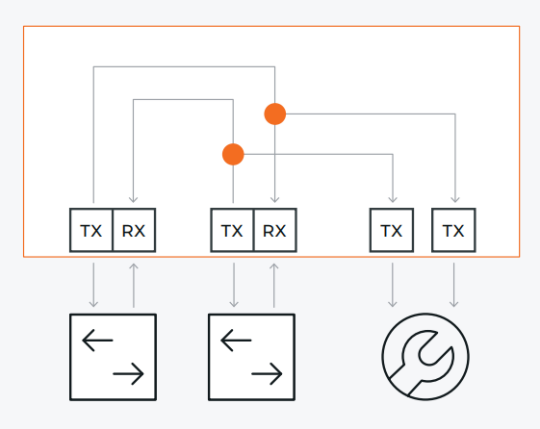



Figure 2.5: Active TAP functioning [8]

As seen in the picture, every single packet gets copied from its origin, and sent out to the monitoring tool, while the original packet continues its path. This is a very useful tool for network monitoring, as it allows the network administrator to have a full copy of the network traffic, and analyze it in real-time, without affecting the original traffic.

Manufacturers claim, that TAP is a better solution for traffic monitoring that Switched Port Analyzer (SPAN), due to the following reasons:

- **Zero packet loss:** TAPs provide a full copy of the network traffic, ensuring that no packets are dropped or lost during monitoring. TAP as well ensures that even faulty packets get copied, providing much more information for IT-Analysts.
- **Complete visibility:** TAPs capture all network traffic, including layer 1 and layer 2 errors, which may not be visible in a SPAN port.
- **Non-intrusive monitoring:** TAPs operate passively and do not interfere with the network, making them ideal for monitoring critical links without introducing latency or packet loss.
- **Security and compliance:** TAPs provide a secure and reliable way to monitor network traffic, ensuring compliance with security and regulatory requirements.

However, in real-world data networks (like 5G networks), the use of SPAN or traffic mirror techniques in general is widely spread, since the huge amount of traffic provides already a decent amount of data to analyze and the use of TAPs is not always possible due to the physical constraints of the network.

### 2.5.5 Virtual Test Access Point (vTAP)

As the name suggests, a Virtual Traffic Access Point (vTAP) is a software-based solution that replicates the functionality of a physical TAP in a virtualized or cloud environment. vTAPs are designed to capture, monitor, and analyze network traffic within virtualized infrastructure, providing visibility into virtual machines, containers, and cloud-based workloads.

vTAPs operate by intercepting network traffic at the virtual network interface level, copying packets, and forwarding them to monitoring tools or security appliances. By replicating the functionality of a physical TAP in a virtual environment, vTAPs enable administrators to monitor network traffic.

A very well known manufacturer as *Gigamon Inc.* [9] provides vTAP alternatives for cloud-based solutions, offering a managed solution leveraging the user to focus on monitoring tasks.

The managed solution from *Gigamon Inc.* is called *GigaVUE Cloud Suite* and it provides a comprehensive and seamless approach to mirror Kubernetes traffic, as well as other cloud-based solutions, to the monitoring tools.

However, it is a paid solution, and for the purpose of this project, a custom vTAP will be developed, leveraging the Kubernetes environment and the tools described in the previous sections.

## 2.6 Virtualized 5G Networks

Virtualized 5G networks are a key enabler of next-generation telecommunications services, offering increased flexibility, scalability, and efficiency compared to traditional network architectures. By leveraging virtualization technologies, such as Network Functions Virtualization (NFV) and Software-Defined Networking (SDN), virtualized 5G networks enable service providers to deliver innovative services and applications with enhanced performance and agility.

Virtualized 5G networks are designed to support a wide range of use cases, including enhanced mobile broadband, massive machine-type communications, and ultra-reliable low-latency communications. These use cases require diverse network capabilities, such as high bandwidth, low latency, and network slicing, which virtualized 5G networks can provide through dynamic resource allocation and service orchestration.

In this section, the concepts of most widely used tools will be introduced, so there is enough theoretical framework that can be studied and chosen in the following chapters to simulate a 5G network and provide the necessary data for the network traffic monitoring and tapping.

### 2.6.1 Open5Gs

Open5GS is a modern software implementation of 5G Core and EPC [14], providing a comprehensive suite of functions that support the 4G and 5G core network infrastructure. The project adhered to the 3GPP Release 15 at the beginning, although it now provides Release 17 compliance as well (from Release `v2.6.1` on March 2023[2]), although in its latest standards and offers an open-source approach to deploying a 5G core network. Open5GS is designed to provide a functional and testing platform that can be used by researchers, developers, and telecommunications providers to innovate and test new 5G-oriented services and applications.

[2]Github Release Notes v2.6.1 https://github.com/open5gs/open5gs/releases/tag/v2.6.1

### Architecture of Open5GS

The Open5GS framework consists of several components, each corresponding to a specific functional entity within the 4G LTE and 5G networks, like AMF, SMF, UPF, UDM, PCF, and other essential functions as explained in section  section 2.1.2.

A complete architecture of how these components interact with each other is shown in the following figure:

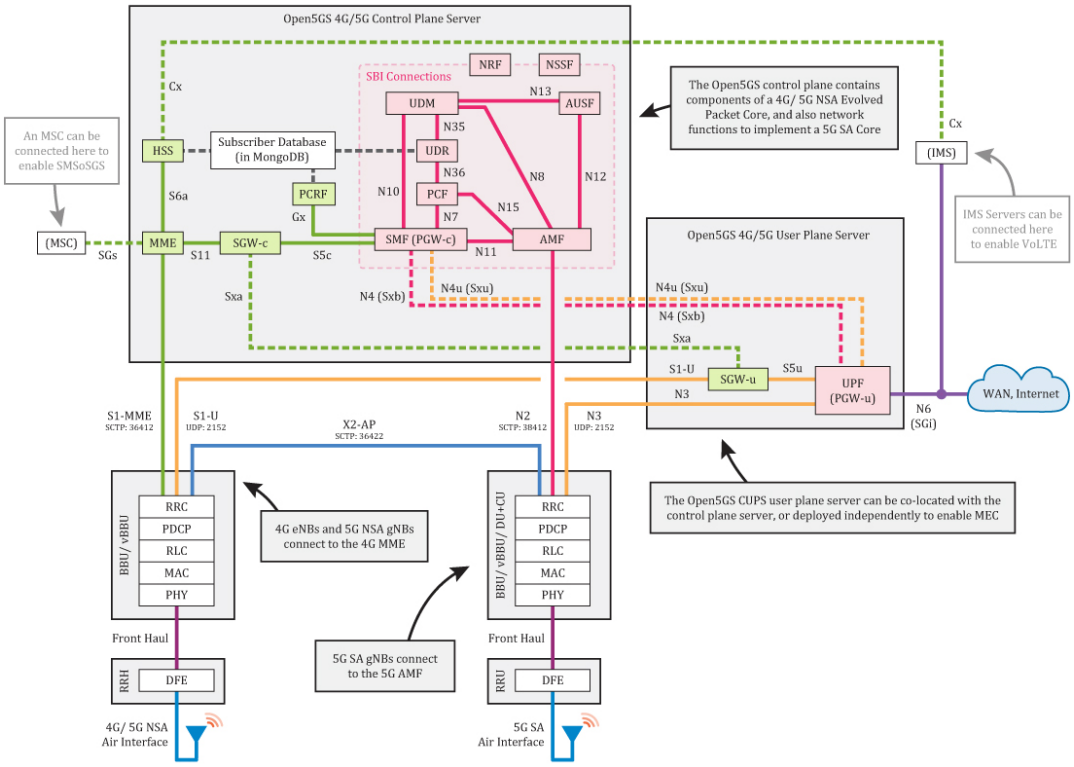



Figure 2.6: Open5GS Architecture [14]

As seen in the picture these elements are interconnected via standardized interfaces, facilitating a modular and scalable 5G core network system. The study of this kind of interfaces and how they make communication between network functions possible, is out of the scope of the current project, therefore they will not be covered.

### Open5GS set of features

Open5GS provides a comprehensive set of features that enable the deployment of a functional 5G core network. Some of the key features include:

- PDU session establishment and session modification.
- Integration with multiple external systems, giving the possibility to deploy a complete 5G network. To mention a few of them:
  - *IMS (IP Multimedia Subsystem):* for delivering multimedia services.
  - Existing LTE networks for NSA (Non-Standalone) deployments where 5G enhancements are provided alongside 4G capabilities.
  - *SDN (Software Defined Networking) and NFV (Network Functions Virtualization):* technologies for enhanced network management and orchestration.
  - Various virtualized RAN options like UERANSIM, srsRAN and others.
- Embedded 5G roaming implementation.
- Support for network slicing, enabling the creation of isolated virtual networks tailored to specific use cases.
- IPv6 support, Support of USIM cards using Milenage confidentiality algorithm, QoS implementation and more.

Inside the official Open5GS documentation[3], there is already a lot of information about how to deploy and configure the software. Given the objectives of the present thesis, we will pay special attention to the deployment using `Helm` for a Kubernetes cluster, which will be used and explained in the next chapter.

Open5GS has reached a clear maturity as an open source project as well, being sponsored by emerging connectivity providers and supported by a large community of developers and users. The project is actively maintained and updated, with new features and improvements being added regularly responding to updates on 3GPP releases and community feedback.

[3] https://open5gs.org/open5gs/docs/

### 2.6.2 Free5GC

free5GC is an open-source initiative that also aligns with the 3rd Generation Partnership Project (3GPP) standards [18] (supported up until Release 15 according to documentation [5]), providing a comprehensive toolkit for deploying a standalone 5G core network that is adaptable, scalable, and forward-compatible.

#### Architecture of Free5GC

There is no actual architecture diagram provided in the official documentation, but the components are similar to the ones provided by Open5GS, as they are based on the same 3GPP standards, so all the network functions specified in 5G release 15 are duly provided on Free5GC.

#### Free5GC set of features

free5GC offers a decent amount of features, some of them are:

- Provides the complete set for a 5G core network, including the AMF, SMF, UPF, UDM, PCF, and other essential functions, does not provide 4G network functions.
- Network Slicing support, allowing the creation of isolated virtual networks tailored to specific use cases.
- Multiple UPF support, enabling the deployment of multiple UPFs to handle different traffic types and requirements.
- OAuth 2.0 authentication and authorization support, ensuring secure connection for new network functions.

The `Free5GC` project is sponsored by the National Chiao Tung University in Taiwan, and it is actively maintained and updated by a team of developers. The project has a growing community of users and contributors, and it is well-documented, making it an accessible and reliable option for deploying a 5G core network. It has also official tutorials to deploy a 5G core in Kubernetes clusters using tools like `Helm` and `Microk8s`, although the community is not that much engaged and new releases are slowly being published.

### 2.6.3 UERANSIM

UERANSIM is an open-source 5G UE (User Equipment) simulator that provides a virtualized environment for testing and developing 5G networks [19]. It allows users to simulate 5G UEs and interact with 5G core networks, enabling testing of various 5G features and functionalities. UERANSIM is designed to be lightweight, easy to deploy, and highly configurable, making it an ideal tool for researchers, developers, and network operators working with 5G technologies.

UERANSIM is a very minimalistic tool, as it only provides two technologies:

- **UE Simulation:** UERANSIM simulates 5G UEs as Standalone (3GPP Access), allowing users to create virtual UEs with specific configurations and behaviors. Users can define parameters such as the UE's identity, capabilities, and connection settings, enabling detailed testing and analysis of 5G network interactions.
- **5G Standalone RAN (CU gNB):** Provides a fully functional gNodeB in order to register UEs and establish a connection with the 5G core network. This allows users to emulate the end-to-end functionality of a 5G network, including registration, authentication, and data transfer.

Along this technologies, `UERANSIM` provides the following interfaces to interact with the 5G core network:

- **Control Interface (between RAN and AMF):** It provides the user two main functional layers which are:
  - *Non-Access Stratum (NAS) Layer:* It is responsible for the signaling between the UE and the AMF. It is used to establish and release connections, perform registration, and handle mobility events.
  - *Next Generation Application Protocol (NGAP) Layer:* It is responsible for the signaling between the RAN and the AMF. It is used to establish and release connections, perform registration, and handle mobility events.
- **User Plane Interface (between RAN and UPF):** It is responsible for the data transfer between the UE and the UPF. It is used to transmit user data packets between the UE and the UPF. It implements GPRS Tunneling Protocol (GTP) protocol for the data transfer.

### 2.6.4 srsRAN

The srsRAN Project is a complete 5G RAN solution, featuring an ORAN-native CU/DU developed by SRS. The solution includes a complete L1/2/3 implementation with minimal external dependencies. Portable across processor architectures, the software has been optimized for x86 and ARM. srsRAN follows the 3GPP 5G system architecture implementing the functional splits between Distributed Unit (DU) and Centralized Unit (CU). The CU is further disaggregated into control plane (CU-CP) and user-plane (CU-UP).

srsRAN Project is a 5G CU/DU solution and does not include a UE application. However, srsRAN 4G does include a prototype 5G UE (srsUE) which can be used for testing. This application note shows how to create an end-to-end fully open-source 5G network with srsUE, the srsRAN Project gNodeB and Open5GS 5G core network.

#### srsRAN features

The srsRAN Project provides 5G RAN solution, including (among others) the following features:

- 3GPP release 17 aligned.
- FDD/TDD supported, all FR1 bands.
- Network Slicing.
- All physical channels including PUCCH Format 1 and 2, excluding Sounding-RS

As seen in the feature set, the srsRAN project aims to target users and researches that are more interested into the physical layer of the 5G-RAN network portion, but sadly it does not provide yet a fully functional 5G compliant User Equipment (UE).

## 2.7 Summary

This chapter provided an overview of the key concepts and technologies that form the foundation of the present thesis. It introduced Kubernetes, Custom Resource Definitions, Custom Controllers, Helm, Network Monitoring, Virtualized 5G Networks, Open5GS, and UERANSIM. These technologies and concepts are essential for understanding the subsequent chapters, which focus on deploying a virtual Traffic Access Point (vTAP) in a Kubernetes environment, monitoring network traffic, and performing forensic analysis on 5G networks.

Kubernetes is a powerful container orchestration platform that enables the deployment and management of containerized applications. Custom Resource Definitions and Custom Controllers extend Kubernetes' capabilities by allowing users to define custom resources and automate complex workflows. Helm simplifies the deployment of Kubernetes applications by providing a templating engine and package management system. Network monitoring is crucial for ensuring the performance, security, and reliability of networked systems. Virtualized 5G networks leverage NFV and SDN technologies to deliver innovative services and applications with enhanced performance and agility. Open5GS and UERANSIM are open-source tools for deploying and simulating 5G core networks, providing a comprehensive suite of functions for testing and developing 5G services.

The following chapters will build on these concepts and technologies to deploy a vTAP in a Kubernetes environment, monitor network traffic, and perform forensic analysis on 5G networks. The practical implementation of these tasks will demonstrate how Kubernetes, Open5GS, UERANSIM, and other tools can be used to create a comprehensive network monitoring and forensic analysis solution for 5G networks.

# 3 Testbed Design and Implementation

## 3.1 Introduction

In this chapter, the design and implementation of the testbed used to evaluate the performance of the proposed algorithms is presented.

The main goal is to provide an environment that allows easy experimentation, encompassing a 5G core network and a RAN portion as well, so the further options to capture and monitor traffic are easy to test and evaluate. The testbed was designed as well to allow the experimenters to easily scale and deploy multiple clusters at the same time, so more than one tests can be carried out at the same time.

## 3.2 Overview of the Technology Stack

In this section, the technology stack which was chosen will be presented, as well as the reasons behind the choices made.

### 3.2.1 Kubernetes

One could say that Kubernetes is the *de facto* standard for container orchestration, and the rich set of features and concepts has been already described in section section 2.3.

Considering that the use and possible development that can be implemented on top of Kubernetes for any kind of distributed system, it was chosen as the main platform to deploy the 5G core network and the RAN portion. The wide support given by both the cloud-native and the *telecommunications* industry makes Kubernetes the right choice for this project. Since there is multiple tools already developed by multiple providers that are already *containerized*.

There exist another alternatives to Kubernetes, like Docker Swarm, Mesos, or Nomad, but Kubernetes is the most mature and feature-rich of them all and offers a broader spectre of tooling that can be useful for future projects or developments. A table for the most used container orchestration platforms is presented in appendix .1.1 The

table just solidifies the thought, that Kubernetes is the best choice, since the other alternatives bring along more requirements like the need of a Zookeeper cluster for Mesos, or the need of a Consul cluster for Nomad. Kubernetes is the most mature and feature-rich of them all and offers a broader spectre of tooling that can be useful for future projects or developments.

Important mention as well, is that the Kubernetes version used on the project is *1.27.1*, which is the latest stable version at the time of writing.

### 3.2.2 Calico as Container Network Interface (CNI)

The Container Network Interface CNI chosen for the project is Calico. Calico is an open-source networking and network security solution for containers, virtual machines, and native host-based workloads. Calico provides a highly scalable networking and network policy solution for connecting Kubernetes pods based on the same industry-leading technology that powers some of the world's largest and most secure networks. There are some other excellent alternatives, like Cilium or Flannel. A comparison with the other alternatives is provided in appendix .1.1

This comparison analysis was carried out actually with a pure research and theoretical approach [17] [4], not from a practical test since it is not one of the goal of the projects.

However, it is worth mentioning that at the final phase of the practical work of the thesis, there was not still a CNI, based on *extended Berkeley Packet Filter* (eBPF) like Cilium, that provides *full support* for Secure Control Transmission Protocol (SCTP) in the Kubernetes cluster [2]. This is a feature that is needed for the 5G core network to work properly, since the SCTP is the protocol used for the communication between the AMF and the SMF (among others), and therefore it can't be used in a 5G core working on a Kubernetes cluster.

### 3.2.3 Container Runtime Interface

There is a great amount of options for CRI in the cloud-native community, being the most used and well adopted containerd, CRI-O and docker (in that order).

A comparison in the most important features is provided in the following table in appendix .1.1.

The Container Runtime Interface CRI chosen for this project is *CRI-O*. CRI-O is an implementation of the Kubernetes Container Runtime Interface (CRI) to enable using Open Container Initiative (OCI) compatible runtimes. Although the mostly wide

adopted option is *containerd*. The main reason for choosing CRI-O is that it is a much lightweight alternative that is very stable and provide enough robustness for a 5G core network to run on top of it. It is also one of the CRI options that is supported by one of the virtual TAP alternatives to be studied in future chapters (kokotap, to be described in chapter 4)

### 3.2.4 5G Simulation Tools

The 5G simulation tools picked for this master thesis project are *Open5GS* and *UERANSIM*.

#### Open5GS for 5G Core Network

As described in sections section 2.6, both Open5GS and Free5GC (which can be considered as the most adopted 5G implementations in the industry) provide a complete 5G network architecture, being both a Open5GS is a complete open-source implementation of the 5G core network. And a comparison table is provided as well in appendix .1.1.

As described also in sections section 2.6.1 and section 2.6.2, the set of features is wide for both tools, but Open5GS was chosen because it is more user-friendly, provides great integration with most of the virtualized RAN simulation tools, and is more suitable for educational and research purposes, such the current project objectives.

### 3.2.5 UERANSIM for RAN Simulation

As explained in sections section 2.6.3 and section 2.6.4, UERANSIM provides a much more lightweight solution to simulate the RAN portion of the 5G network. It is a great tool for testing and validating the 5G core network, and it is also very easy to use and deploy. It is also the most used tool for the RAN simulation in the industry, and it is also the most suitable for the current project objectives.

Since the actual purpose is to capture and mirror simulated user traffic, UERANSIM is the most suitable option for this project.

## 3.3 Kubernetes Cluster Design

### Cluster features

The testbed cluster was deployed in the Data Network Laboratory from the University of Applied Sciences of Cologne with three virtual machines as follows:

- *Control Plane Node:* A single control plane node, 8 vCPUs, 32GB RAM, 150GB SSD.
- *Worker Node 1 and 2:* Two worker nodes, each with 4 vCPUs, 16GB RAM, 100GB SSD.
- *Collection endpoint:* One additional endpoint, which will not be included in the cluster but will be used to collect the mirror traffic and further remote testing.

  The Collection endpoint is a VM with 4 vCPUs, 16GB RAM, 200GB SSD.

The chosen VM settings were chosen by oversizing the resources, so the cluster can be used for multiple tests at the same time, and the performance of the cluster is not affected by the resources. The cluster was deployed using the following software versions:

- *Operating System:* Ubuntu 20.04 LTS.
- *Kubernetes Version:* 1.27.1.
- *Container Runtime:* CRI-O v1.26.4
- *Container Network Interface:* Calico v3.21.0
- *Helm:* v3.11.0
- *Open5gs*: version 2.7.0
- *UERANSIM*: version 3.2.6

All software versions were the latest at the moment of installation.

## 3.4 Deployment of the 5G Core and RAN in Kubernetes

The deployment of the 5G core network and the RAN portion was accomplished using Helm Charts, which were developed and provided by Gradiant[1] and include all the necessary components to deploy the 5G core network and the RAN portion with UREANSIM in a Kubernetes cluster. The rollout is pretty straightforward, and can be accomplished with a single `helm install command as shown below:`

Listing 3.1: Helm installation for Open5GS and UERANSIM

```
andres@k8s-cp:~$ helm install open5gs --namespace open5gs oci://registry-1.
    docker.io/gradiant/open5gs --version 2.2.0 --values https://gradiant.
    github.io/5g-charts/docs/open5gs-ueransim-gnb/5gSA-values.yaml

Pulled: registry-1.docker.io/gradiant/open5gs:2.2.0
Digest: sha256:99d49ab6c78be31dd2c3a99a0780de79bd22d0bfa9df734ca270594
NAME: open5gs
LAST DEPLOYED: Fri Nov  10 10:44:10 2023
NAMESPACE: open5gs
STATUS: deployed
REVISION: 1
TEST SUITE: None

--------

andres@k8s-cp:~$ helm install ueransim-gnb -n open5gs oci://registry-1.docker.
    io/gradiant/ueransim-gnb --version 0.2.6 --values https://gradiant.github.
    io/5g-charts/docs/open5gs-ueransim-gnb/gnb-ues-values.yaml

Pulled: registry-1.docker.io/gradiant/ueransim-gnb:0.2.6
Digest: sha256:feedcd66907e921775fc29065a54197c1ef66b13d62a6cb4cccbd42
NAME: ueransim-gnb
LAST DEPLOYED: Fri Nov  10 10:45:55 2023
NAMESPACE: open5gs
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
ueransim-gnb successfully installed!
Check gnodeb log with:

‘‘‘
kubectl -n open5gs logs deployment/ueransim-gnb
‘‘‘

---

You have also deployed 2 ues. You can enter ues terminal with:

‘‘‘
kubectl -n open5gs exec -ti deployment/ueransim-gnb-ues -- /bin/bash
‘‘‘
There is a tun interface for each ue.
```

[1] https://www.gradiant.org/en/about/

```
You can bind your application to the interface to test ue connectivity.
Example:

'''
ping -I uesimtun0 gradiant.org
traceroute -i uesimtun0 gradiant.org
curl --interface uesimtun0 https://www.gradiant.org/
'''

You can also deploy more ues connected to this gnodeb with gradiant/ueransim-
    ues chart:

'''
helm install -n open5gs ueransim-ues gradiant/ueransim-ues --set gnb.hostname=
    ueransim-gnb
```

As seen in the snippets above, `Gradiant` already includes some values to facilitate a faster and ready-for-testing deployment of a complete 5G network. The values are provided in the Helm Charts repository are included in the Attachments section of this document.

Is worth mentioning that there is a special *Kubernetes Deployment* included in the helm chart called *"populate"*, which acts as a helper to initiate the MongoDB included in Open5Gs, to include two users in the database, and to create the necessary configuration files for the UERANSIM to connect to the Open5GS. This function has the following setup, which is extracted by running the `-dry-run` option during the `helm installation`:

Listing 3.2: Populate deployment from Helm Chart

```
# Source: open5gs/templates/populate-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: open5gs-populate
    namespace: "open5gs"
    labels:
spec:
    template:
    metadata:
    spec:
        initContainers:
        - name: init
        image: docker.io/gradiant/open5gs-dbctl:0.10.3
        imagePullPolicy:
        env:
        - name: DB_URI
            value: mongodb://open5gs-mongodb/open5gs
        command:
        - /bin/bash
        - -c
        - |
            "open5gs-dbctl add_ue_with_slice "!!IMSI1" 465B5CE8B199B49
                E8ED289DEBA94 internet 1 111111 \
```

```
            open5gs-dbctl add_ue_with_slice "!!IMSI2" 465B5CE8B199B48
                E8ED289DEBA95 internet 1 111111"
        containers:
        - name: populate
        image: docker.io/gradiant/open5gs-dbctl:0.10.3
        imagePullPolicy:
        env:
        - name: DB_URI
            value: mongodb://open5gs-mongodb/open5gs
        command:
        - /bin/bash
        - -c
        - "tail -f /dev/null"
---
```

The `populate` function is a simple `initContainer` that runs the `open5gs-dbctl` command to add two users to the MongoDB database, which are used by the UERANSIM to connect to the Open5GS.

## 3.5 Testing and Validation

As show in 3.1, the `UERANSIM` installation already provides some verification commands, to check if the deployment is successful and the Open5GS core is working as expected and the UEs are registered and connected to the gNB. Some test are shown below:

Listing 3.3: Testing the Open5GS and UERANSIM deployment

```
## Check gNB and UE registration
andres@k8s-cp:~$ kubectl -n open5gs logs deployment/ueransim-gnb
N2_BIND_IP: 10.42.5.48
N3_BIND_IP: 10.42.5.48
RADIO_BIND_IP: 10.42.5.48
AMF_IP: 10.105.163.52
Launching gnb: nr-gnb -c gnb.yaml
UERANSIM v3.2.6
[2024-06-08 11:31:05.029] [sctp] [info] Trying to establish SCTP connection...
     (10.103.83.127:38412)
[2024-06-08 11:31:05.038] [sctp] [info] SCTP connection established
    (10.103.83.127:38412)
[2024-06-08 11:31:05.039] [sctp] [debug] SCTP association setup ascId[14]
[2024-06-08 11:31:05.040] [ngap] [debug] Sending NG Setup Request
[2024-06-08 11:31:05.049] [ngap] [debug] NG Setup Response received
[2024-06-08 11:31:05.049] [ngap] [info] NG Setup procedure is successful
[2024-06-08 11:31:06.492] [rrc] [debug] UE[1] new signal detected
[2024-06-08 11:31:06.492] [rrc] [debug] UE[2] new signal detected
[2024-06-08 11:31:06.495] [rrc] [info] RRC Setup for UE[1]
[2024-06-08 11:31:06.495] [rrc] [info] RRC Setup for UE[2]
[2024-06-08 11:31:06.495] [ngap] [debug] Initial NAS message received from UE
    [2]
[2024-06-08 11:31:06.495] [ngap] [debug] Initial NAS message received from UE
    [1]
```

```
[2024-06-08 11:31:06.630] [ngap] [debug] Initial Context Setup Request
    received
[2024-06-08 11:31:06.665] [ngap] [debug] Initial Context Setup Request
    received
[2024-06-08 11:31:06.725] [ngap] [info] PDU session resource(s) setup for UE
    [2] count[1]
[2024-06-08 11:31:06.756] [ngap] [info] PDU session resource(s) setup for UE
    [1] count[1]

## Connectivity test from UEs to the internet

andres@k8s-cp:~$ kubectl -n open5gs exec -ti deployment/ueransim-gnb-ues -- /
    bin/bash
bash-5.1# ping -I uesimtun0 www.th-koeln.de
PING www.th-koeln.de (139.6.10.199): 56 data bytes
64 bytes from 139.6.10.199: seq=0 ttl=251 time=6.042 ms
64 bytes from 139.6.10.199: seq=1 ttl=251 time=4.076 ms
64 bytes from 139.6.10.199: seq=2 ttl=251 time=4.248 ms
64 bytes from 139.6.10.199: seq=3 ttl=251 time=3.762 ms
64 bytes from 139.6.10.199: seq=4 ttl=251 time=8.056 ms
64 bytes from 139.6.10.199: seq=5 ttl=251 time=4.037 ms
64 bytes from 139.6.10.199: seq=6 ttl=251 time=3.666 ms
64 bytes from 139.6.10.199: seq=7 ttl=251 time=3.671 ms
^C
--- www.th-koeln.de ping statistics ---
8 packets transmitted, 8 packets received, 0% packet loss
round-trip min/avg/max = 3.666/4.694/8.056 ms


## cURL test

bash-5.1# curl --interface uesimtun1 -svo /dev/null https://www.fastly.com/
*   Trying 146.75.117.57:443...
* Connected to www.fastly.com (146.75.117.57) port 443 (#0)
* ALPN: offers h2
* ALPN: offers http/1.1
*  CAfile: /etc/ssl/certs/ca-certificates.crt
*  CApath: none
} [5 bytes data]
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
} [512 bytes data]
* TLSv1.3 (IN), TLS handshake, Server hello (2):
{ [122 bytes data]
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
{ [19 bytes data]
* TLSv1.3 (IN), TLS handshake, Certificate (11):
{ [2537 bytes data]
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
{ [264 bytes data]
* TLSv1.3 (IN), TLS handshake, Finished (20):
{ [36 bytes data]
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
} [1 bytes data]
* TLSv1.3 (OUT), TLS handshake, Finished (20):
} [36 bytes data]
* SSL connection using TLSv1.3 / TLS_AES_128_GCM_SHA256
```

With the tests above, one can conclude that the Open5GS core network is working as expected, and the UEs are connected to the gNB and can access the internet. The next step is to capture the traffic from the UEs and mirror it to the collection endpoint.

# 4 Proposed Solutions

After the previous analysis of several In this chapter, the following solutions which were picked as the best fit for the problem are presented.

- **Kokotap as CLI tool**
- **Istio Service Mesh + Kiali**
- **Kokotap as Kubernetes Operator**

These solutions are described in detail, and further analysis will be introduced in the next chapter.

## 4.1 Kokotap as CLI tool

The first solution is to use the standard, out-of-the-box Kokotap, which works as a command-line interface (CLI) tool. According to its documentation, "*kokotap* provides network tapping for Kubernetes Pod. kokotap creates VxLAN interface to target Pod/Container then do packet mirroring to the VxLAN interface by tc-mirred. kokotap can also create VxLAN interface to Kubernetes target node (e.g. 'kube-master') to capture the traffic or you can specify specific IP addresses for non Kubernetes node for capture." It is a tool initially developed by Red Hat, while pursuing early improvements and monitoring tools to an emerging orchestration tool like Kubernetes was at that time (around year 2018).

It is written in Go, and it is available as a binary file, which can be downloaded from the official GitHub repository. The tool is easy to use, and it is well documented. The user can create a new tap by running the command `kokotap create`, and then the user can specify the target pod or node, and the target interface. The tool will then create a new VxLAN interface, and it will start mirroring the traffic to that interface. The user can then use any packet capture tool, like Wireshark, to capture the traffic.

The main features that can add value to the purpose of the project are the following:

- Easy to use: as it is a single binary file, it is fairly simple to install, use and even extend the functionalities.

- Choose the target: Kokotap gives the user the chance to send the captured traffic to a specific cluster node, or to an external endpoint (meaning outside of the cluster).
- Useful arguments: the user must provide the following mandatory arguments: target pod, target interface on pod, namespace of the pod, mirror type (ingress, egress or both), destination node and IP address. Some other optional arguments are also available, like the VxLAN ID, the MTU, the source IP address, the source port, interface among others.

  Those arguments scope perfectly the target that an operator would like to monitor, and enables a precise network traffic analysis.
- Single binary file developed using Go: The user could even extend the code to add new features or to fix bugs.

But there are some key disadvantages that shall be pointed out:

- Lack of maintenance and community: the last commit on the GitHub repository was made in 2018, and there are no signs of recent updates or maintenance. This has led into some compatibility issues with latest Kubernetes versions, that were met during the research phase.

  There is no visible sign of a community around the project, which means new features or bug fixes are unlikely to be handled, so the tool would eventually become obsolete.
- Poor documentation: the documentation is not very detailed and even inaccurate in some parts, which prevents a new user from understanding how the tool works.
- Limited Container Runtime Interface support: The broader Kubernetes community is moving towards containerd as the default CRI, and kokotap does not support it. (Kokotap only supports Docker and CRI-O).
- Security concerns: kokotap requires the user to handout the kubeconfig file, which contains sensitive information about the Kubernetes cluster like certificates, context information and more. This is an important security concern, as the user must trust the tool to not misuse the information.

Since one of the requirements is to capture the traffic from a Pod, and to send this traffic to an external endpoint, the first thing is to choose the right arguments for the `kokotap create` command. The arguments that were chosen are the following:

- `-pod=`: the target pod name.
- `-interface`: the target interface on the pod.

- `-namespace`: the namespace of the pod.
- `-mirror-type`: the type of traffic to mirror, the option chosen was `both`.
- `-destination-ip`: the destination IP address.
- `-dest-ip`: the destination IP address (of the external endpoint).
- `-kubeconfig`: the path to the kubeconfig file, so the tool is authorized to manage kubernetes resources. This is specified as optional, but after initial testing and by analyzing the code it was found that it is mandatory.

### 4.1.1 Kokotap way of working

To properly use kokotap out-of-the-box, the user must follow the following steps:

1. Clone their Github repository https://github.com/redhat-nfvpe/kokotap, afterwards the user can either add the `cmd` directory to the `PATH` environment variable, or run the binary file directly from the `kokotap` directory.
2. Run the `./kokotap` command, and provide arguments accordingly.
3. Capture the traffic in the destination, normally using a `tcpdump` command with appropriate filters and usually store it in a `.pcap` file.

What kokotap does in the background for the user is the following:

- Looks fot the provided pod and namespace, and retrieves the container ID and the IP address of the kubernetes node on which the pod is running.
- Dumps a YAML file with the necessary to create a pod, known as *sender* pod, which will be used to mirror the traffic.
- Creates a new pod on the same node, which will be used to mirror the traffic.
- Creates a new VxLAN tunnel interface, with the `vxlanID` provided by the user, and using as source the IP address of the node and as destination the IP address of the external endpoint (provided by the user as well).
- Mirrors the traffic from the pod's interface and sends its through the VxLAN tunnel.

An example of the output of the `kokotap` command is shown below:

Listing 4.1: Output of the kokotap command

```
andres@k8s-cp:~/koko/kokotap ./kokotap --pod=pods_name --vxlan-id=ID --
    mirrortype=both --dest-ip=external_ip --namespace="pod_namespace" --
    kubeconfig="/path/to/kubeconfig_file"
---
apiVersion: v1
kind: Pod
metadata:
  name: kokotap-pods_name-sender
spec:
  hostNetwork: true
  nodeName: k8s-w1.5g.dn.th-koeln.de
  containers:
    - name: kokotap-pods_name-sender
      image: quay.io/s1061123/kokotap:latest
      imagePullPolicy: Always
      command: ["/bin/kokotap_pod"]
      args: ["--procprefix=/host", "mode", "sender", "--containerid=cri-o
          ://6636", #shortened for brevity
             "--mirrortype=both", "--mirrorif=eth0", "--ifname=mirror",
             "--vxlan-egressip=pod_nodes_name", "--vxlan-ip=external_ip", "--
                 vxlan-id=ID",
             "--vxlan-port=4789"]
      securityContext:
        privileged: true
      volumeMounts:
      - name: var-crio
        mountPath: /var/run/crio/crio.sock
      - name: proc
        mountPath: /host/proc
  volumes:
    - name: var-crio
      hostPath:
        path: /var/run/crio/crio.sock
    - name: proc
      hostPath:
        path: /proc
---
```

It is important to point out, that in this case the container runtime being used is CRI-O, and the pod is being created in the same node as the target pod. The user must have the necessary permissions to create pods in the cluster, and the kubeconfig file must be provided to the tool.

Since kokotap only dumps a YAML file to create a new pod, the user can use this output to create a new pod directly, by piping the output to the `kubectl apply -f -` command. This is useful for debugging purposes, and to understand how the tool works.

Listing 4.2: Creating a new pod using the output of kokotap

```
  andres@k8s-cp:~/koko/kokotap ./kokotap --pod=pods_name --vxlan-id=ID --
      mirrortype=both --dest-ip=external_ip --namespace="pod_namespace" --
      kubeconfig="/path/to/kubeconfig_file" | kubectl apply -f -
```

```
pod/kokotap-pods_name-sender created
```

To delete the pod, one just needs to pipe the output to the `kubectl delete -f -` command. This is recommended to do after the user has finished capturing the traffic, as the pod will keep running until it is manually deleted.

Listing 4.3: Deleting a pod using the output of kokotap

```
andres@k8s-cp:~/koko/kokotap ./kokotap --pod=pods_name --vxlan-id=ID --
    mirrortype=both --dest-ip=external_ip --namespace="pod_namespace" --
    kubeconfig="/path/to/kubeconfig_file" | kubectl apply -f -
pod/kokotap-pods_name-sender created
```

As seen in the code snippets, and in the dumped YAML file, a new `sender` pod is created, with the binary `kokotap_pod` running as the main process. This binary is responsible for two main tasks:

1. **Build VXLAN interface**: Creates a new VxLAN interface, with the provided `vxlanID`, and the source IP address of the node and the destination IP address of the external endpoint. This is accomplished by using the `kokotap api` go package which has pre-built functions to create VxLAN interfaces accordingly to the user's passed arguments.
2. **Mirror traffic**: The binary uses the `tc` command to mirror the traffic from the target interface to the VxLAN interface. This is done by using the `tc-mirred` action, which is a Linux kernel module that allows to mirror traffic from one interface to another.

#### Kokotap as CLI tool - Test

The testbed with the inclusion of kokotap looks like the following:

As seen in the figure, one can pick any pod running in the kubernetes cluster to mirror traffic, and kokotap will build the machinery behind to enable capture traffic from the pod selected interface and send it to an external endpoint.

Multiple tests were carried out to verify the functionality of kokotap, and the results were as expected. The tool was able to create a new pod, and to mirror the traffic from the target pod to the VxLAN interface. The user was able to capture the traffic using `tcpdump`, and to analyze it using Wireshark. The tool was able to mirror both ingress and egress traffic, and the user was able to specify the destination IP address.

In the figure below one can see how the `kokotap sender` pod is created:

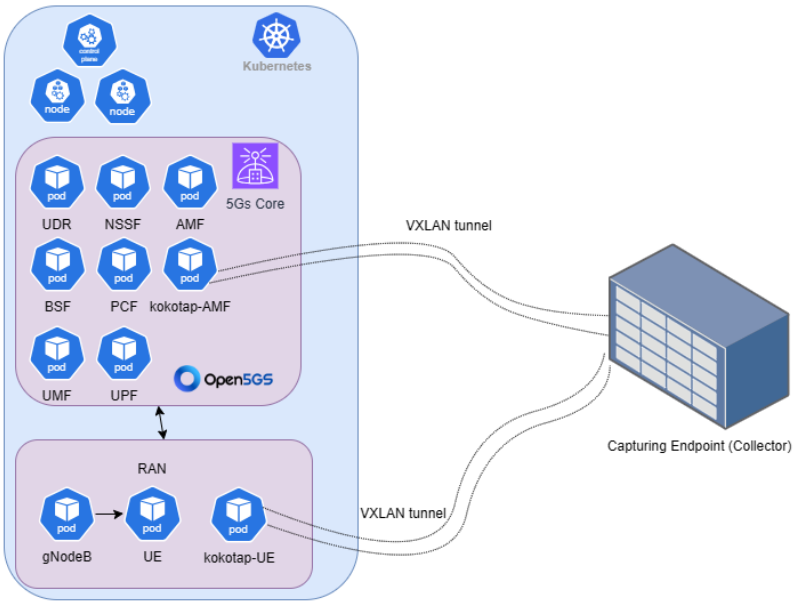



Figure 4.1: Testbed with kokotap

Listing 4.4: Kokotap pod creation

```
andres@k8s-cp:~/koko/kokotap$ kubectl get pod kokotap-ueransim-gnb-ues-6
    c7d5c7bfb-rb4q6-sender -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: ## skipped for breviety
  creationTimestamp: "2024-05-31T07:55:58Z"
  name: kokotap-ueransim-gnb-ues-6c7d5c7bfb-rb4q6-sender
  namespace: default
  resourceVersion: "80379625"
  uid: d6e07e10-30ae-4dfd-9b1a-5f3250c1869b
spec:
  containers:
  - args:
    - --procprefix=/host
    - mode
    - sender
    - --containerid=cri-o://6636da5dc808a # shortened to fit
    - --mirrortype=both
    - --mirrorif=eth0
    - --ifname=mirror
    - --vxlan-egressip=192.168.1.108
    - --vxlan-ip=192.168.1.109
    - --vxlan-id=1100
    - --vxlan-port=4789
    command:
```

```
    - /bin/kokotap_pod
    image: quay.io/s1061123/kokotap:latest
    imagePullPolicy: Always
    name: kokotap-ueransim-gnb-ues-6c7d5c7bfb-rb4q6-sender
    resources: {}
    securityContext:
      privileged: true
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/crio/crio.sock
      name: var-crio
    - mountPath: /host/proc
      name: proc
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-xstsp
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  hostNetwork: true
  nodeName: k8s-w1.5g.dn.th-koeln.de
  preemptionPolicy: PreemptLowerPriority
  priority: 0
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - hostPath:
      path: /var/run/crio/crio.sock
      type: ""
    name: var-crio
  - hostPath:
      path: /proc
      type: ""
    name: proc
  - name: kube-api-access-xstsp
    projected:
      defaultMode: 420
      sources:
      - serviceAccountToken:
          expirationSeconds: 3607
          path: token
      - configMap:
          items:
          - key: ca.crt
            path: ca.crt
```

```
          name: kube-root-ca.crt
      - downwardAPI:
          items:
          - fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
            path: namespace
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2024-05-31T07:55:58Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2024-05-31T07:56:03Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2024-05-31T07:56:03Z"
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2024-05-31T07:55:58Z"
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID: cri-o://b6a9e51d00b0d05acca5f82 # shortened to fit
    image: quay.io/s1061123/kokotap:latest
    imageID: quay.io/s1061123/kokotap@sha256:25d2255dc2b07 # shortened to fit
    name: kokotap-ueransim-gnb-ues-6c7d5c7bfb-rb4q6-sender
    ready: true
    restartCount: 0
    started: true
    state:
      running:
        startedAt: "2024-05-31T07:56:02Z"
  hostIP: 192.168.1.108
  phase: Running
  podIP: 192.168.1.108
  podIPs:
  - ip: 192.168.1.108
  qosClass: BestEffort
  startTime: "2024-05-31T07:55:58Z"
```

The host that is receiving the mirrored traffic must have previously created the VXLAN interface, and the user must have the necessary permissions to create the interface.

The user can use the `ip` command to create the interface, and to set the necessary parameters, as shown below:

Listing 4.5: Setting up the VxLAN interface

```
andres@vtap-endpoint:~$ sudo ip link add vxlan0 type vxlan id 1100 dev ens160
    dstport 4789
andres@vtap-endpoint:~$ sudo ip link set up vxlan0
```

With the `kokotap sender` created and the VXLAN properly created in the receiving host, the packet capture can already begin, by runnning a `tcpdump` command,

**Look into captured pods traffic** Normally, a production 5G cluster would generate a huge amount of traffic itself, but for the sake of this test, the user can log into the pod and generate traffic with a simple `ping` command, and also a `cURL` request to `www.google.com`, as described below:

Listing 4.6: Generating traffic in the pod

```
andres@k8s-cp:~$ kubectl exec -ti pods/ueransim-gnb-ues-6c7d5c7bfb-mgnfp -- /
    bin/bash
bash-5.1#
bash-5.1#
bash-5.1# ip a | grep mirror ## check if the mirror interface is up
126: mirror@if126: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
    state UNKNOWN group default qlen 1000
bash-5.1#
bash-5.1#
bash-5.1# ping 4.2.2.2 -I uesimtun0 # generate ICMP traffic with ping command
PING 4.2.2.2 (4.2.2.2): 56 data bytes
64 bytes from 4.2.2.2: seq=0 ttl=55 time=10.743 ms
64 bytes from 4.2.2.2: seq=1 ttl=55 time=9.947 ms
64 bytes from 4.2.2.2: seq=2 ttl=55 time=9.905 ms
64 bytes from 4.2.2.2: seq=3 ttl=55 time=9.767 ms
64 bytes from 4.2.2.2: seq=4 ttl=55 time=10.080 ms
64 bytes from 4.2.2.2: seq=5 ttl=55 time=9.856 ms
64 bytes from 4.2.2.2: seq=6 ttl=55 time=14.076 ms
64 bytes from 4.2.2.2: seq=7 ttl=55 time=24.281 ms
64 bytes from 4.2.2.2: seq=8 ttl=55 time=11.874 ms
64 bytes from 4.2.2.2: seq=9 ttl=55 time=15.944 ms
64 bytes from 4.2.2.2: seq=10 ttl=55 time=9.861 ms
64 bytes from 4.2.2.2: seq=11 ttl=55 time=11.487 ms
64 bytes from 4.2.2.2: seq=12 ttl=55 time=9.460 ms
^C
--- 4.2.2.2 ping statistics ---
13 packets transmitted, 13 packets received, 0% packet loss
round-trip min/avg/max = 9.460/12.098/24.281 ms

bash-5.1# curl www.google.com --interface uesimtun0
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="
    de">
## output skipped for brevity
```

```
&ei="+b+"&tgtved="+f+"&jsname="+(a||"")))}}else I=a,H=[b]}window.document.
    addEventListener("DOMContentLoaded",function(){document.body.
    addEventListener("click",J)});}).call(this);</script></body></html>
bash-5.1#
```

The user can use the `tcpdump` command to capture the traffic already in the external host, this is shown below:

Listing 4.7: Capturing the traffic in the external host

```
$sudo tcpdump -vvvi vxlan0 -s 65535 -w kokotap-cli-test.pcap
tcpdump: listening on vxlan0, link-type EN10MB (Ethernet), capture size 65535
    bytes
^C171 packets captured
175 packets received by filter
0 packets dropped by kernel
```

The user can then analyze the captured traffic using Wireshark, and the results should be similar to the ones shown in the figure below. For the sake of proving the test, a filter was applied to only show GTP traffic, which is the protocol used in 5G networks to encapsulate user data.

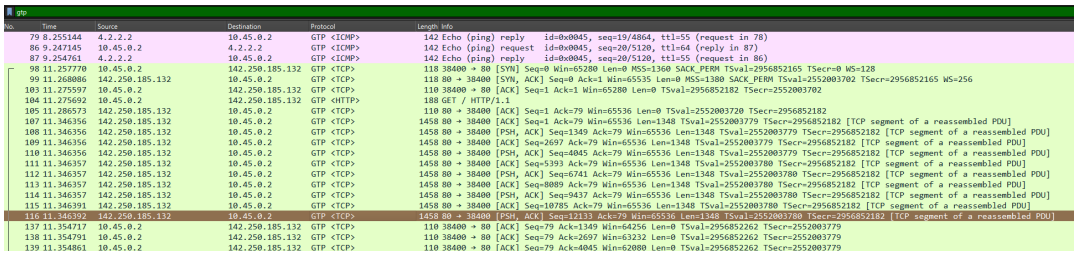



Figure 4.2: Wireshark capture of the traffic, filtered to show only GTP

The test as shown was successful, the objective of sending "live" traffic to an external endpoint was duly accomplished.

## 4.2 Extending Kubernetes API with Kokotap CRD

The second solution is to use Kokotap as a Kubernetes Operator. As explained in section 2.3, Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. Operators follow Kubernetes principles, notably the control loop.

### 4.2.1 Motivation for Kokotap CRD

The current state of the art in Kubernetes is to use Operators to manage applications and their components. As seen in the previous section 2.3.3, Custom Resource Definitions (CRDs) are used to extend the Kubernetes API, and to create new resources. Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. Operators follow Kubernetes principles, notably the control loop.

As described in the previous section "insertarreferencia", kokotap comes as a CLI tool which highly relies on the user installing go in its local system (or in the system where the tool is going to be used). This is a limitation, as the user must have the necessary permissions to install software in the system, and to run the tool. It is also an external tool not embedded into kubernetes, wich also poses as a disadvantage taking out the flexibility of kuberentes to provide easy access for user to their resources that they want to build.

It would be much better for a kubernetes administrator (or anyone on an IT-team, that must manage kubernetes clusters and capture traffic as this project requests) to have a tool that is in-band-managed by kubernetes itself, even most commmonly, by pure and easy `kubectl` commands. This is where the idea of extending the kubernetes API with a CRD comes in.

#### Designing the Kubernetes' CRD and Custom Operator

The process of designing a custom resource definition (CRD) requires a clear understanding of the problem that the CRD is trying to solve, and the resources that the CRD is going to manage. In this case, the CRD is going to manage the resources that are necessary to capture the traffic from a pod, and to send it to an external endpoint.

A very practical approach was taken to design the CRD, and it is described as follows:

- **Extend Kokotap as a Custom Resource**: As analyzed before, `kokotap` creates a pod which contains a binary that handles all the logic behind the traffic mirroring. Since there is already a container image built with the `kokotap_pod` binary, the idea is to create this pod on a more automated fashion, with a custom kubernetes controller and manage it with a custom resource.
- **Define the Custom Resource**: The custom resource must contain all the necessary information to create the `kokotap_pod` pod. This includes the target pod, the target interface, the mirror type, the destination IP address, the

destination node, the VxLAN ID, the MTU, the source IP address, the source port, and the interface. The custom resource must also contain the status of the pod, and the status of the VxLAN interface.

- **Define the Custom Controller**: The custom controller must watch for changes in the custom resource, and it must create the `kokotap_pod` pod accordingly. By handling out this responsability to a controller, the user can easily create, update and delete the custom resource, and the controller will take care of creating, updating and deleting the pod.

Since all the necessary information is already provided by the `kokotap` tool, the custom resource can be designed to contain all the necessary information to create the pod. The custom resource can be defined as follows:

### Implementing Kokotap CRD

There is a few tools that are recommended to be used when creating a new CRD in kubernetes. The most used in the community are:

- **Kubebuilder**: A framework for building Kubernetes APIs using custom resource definitions (CRDs). It provides a way to define the API, the controller, and the webhook in a single project. It is a very powerful tool, but it is also complex and it has a steep learning curve.
- **Operator SDK**: A framework that uses Kubebuilder under the hood, but it provides a simpler way to create a new CRD. It is a very powerful tool, and it is easier to use than Kubebuilder.
- **Kustomize**: A tool that provides a way to customize Kubernetes resources using patches. It is a very powerful tool, and it is easy to use, but lacks some of the features of Kubebuilder and Operator SDK when creating a new CRD.

The chosen tool was `Operator SDK`, as it provides a much simpler way to create a new CRD, and it is easier to use than Kubebuilder. One can state that Operator SDK is a wrapper around Kubebuilder, as it generates all the necessary files to define a new CRD with the Kubebuilder logic, in a much more intuitive way by taking core of all the scaffolding and boilerplate code that is needed to create a new CRD.

Now another choice needs to be made, as `Operator SDK` offers three options to the user, on which language to use to create the new CRD. The options are:

- **Go**: The most used language to create Kubernetes operators. It is the most powerful language, and it is the most used in the community.

- **Ansible**: A configuration management tool that can be used to create Kubernetes operators. It is a simpler way to create operators, but it is less powerful than Go.
- **Helm**: A package manager for Kubernetes that can be used to create Kubernetes operators. It is a very simple way to create operators, but it is less powerful than Go and Ansible.

From now on, the process of creating the CRD will be described, and the necessary steps to create the custom resource and the custom controller will be shown. All of this based on `Go` programming language.

After installing the Operator SDK, the user can create a new project with the following commands:

Listing 4.8: Creating a new project with Operator SDK

```
$operator-sdk init --domain dn-lab.io --repo github.com/netand593/dn-vtap
Writing kustomize manifests for you to edit...
Writing scaffold for you to edit...
Get controller runtime:
$ go get sigs.k8s.io/controller-runtime@v0.15.0
Update dependencies:
$ go mod tidy
Next: define a resource with:
$ operator-sdk create api
```

The user can then create a new API with the following command:

Listing 4.9: Creating a new API with Operator SDK

```
$operator-sdk create api --group networking --version v1alpha1 --kind Kokotap
    --resource --controller
Writing kustomize manifests for you to edit...
Writing scaffold for you to edit...
api/v1alpha1/kokotap_types.go
api/v1alpha1/groupversion_info.go
internal/controller/suite_test.go
internal/controller/kokotap_controller.go
Update dependencies:
$ go mod tidy
Running make:
$ make generate
mkdir -p /home/andres/az-projects/dn-vtap/bin
test -s /home/andres/az-projects/dn-vtap/bin/controller-gen && /home/andres/az
    -projects/dn-vtap/bin/controller-gen --version | grep -q v0.12.0 || \
GOBIN=/home/andres/az-projects/dn-vtap/bin go install sigs.k8s.io/controller-
    tools/cmd/controller-gen@v0.12.0
/home/andres/az-projects/dn-vtap/bin/controller-gen object:headerFile="hack/
    boilerplate.go.txt" paths="./..."
Next: implement your new API and generate the manifests (e.g. CRDs,CRs) with:
$ make manifests
```

All this previous commands generate the directory structure shown in the picture below:

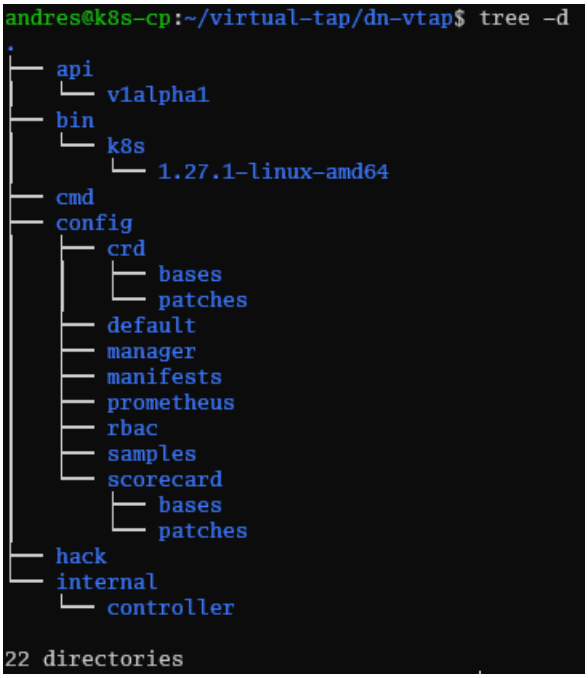



Figure 4.3: Directory structure generated by Operator SDK

This is the directory structure generated by Operator SDK, and it contains all the necessary files to define a new CRD. As seen, most of the scaffolding and boilerplate code is already generated, which leverages the user to focus on the logic behind the CRD, and not on the structure of the project or which files are (or are not) indispensable.

By following the documentation from operator SDK[1] one can continue in a step-by-step fashion to create the custom resource and the custom controller.

Going into details of how the CRD is defined, the following files must be modified to fulfill the objectives:

- `api/v1alpha1/kokotap_types.go`: This file contains the definition of the custom resource, and it must be modified to contain all the necessary information `Spec` and `Status` fields.

[1] https://sdk.operatorframework.io/docs/building-operators/golang/

- `controllers/kokotap_controller.go`: This file contains the logic behind the custom controller, and it must be modified to create the `kokotap_pod` pod accordingly to the custom resource.

The `kokotap_types.go` file will then contain the following `Spec` fields:

Listing 4.10: Kokotap CRD Spec fields

```
  type KokotapSpec struct {
  // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
  // Important: Run "make" to regenerate code after modifying this file

  // Pod's name in which the packet capture should be done
  PodName string `json:"podName"`
  // IP address of the host which will receive the captured packets
  TargetIP string `json:"destIp"`
  // VXLAN ID
  VxLANID int32 `json:"vxlanID"`
  // Namespace of the pod
  Namespace string `json:"namespace"`
  // Type of mirror traffic
  MirrorType string `json:"mirrorType"`
  // Pod Interface to do the tapping (mirror traffic)
  PodInterface string `json:"podInterface"`
  // Image to be used for the kokotap container
  Image string `json:"image"`
  // +kubebuilder:validation:Enum=UDP;TCP
}
```

They are similary tailored as in the previous section section 4.2.1 The comments shown in the code are intentionally left there, as they are result from the scaffolding process from Operator SDK, and they are useful to understand the purpose of each field.

The `Status` fields are also defined in the same file, and they are shown below:

Listing 4.11: Kokotap CRD Status fields

```
  // KokotapStatus defines the observed state of Kokotap
  type KokotapStatus struct {
  // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
  // Important: Run "make" to regenerate code after modifying this file
  Conditions []metav1.Condition `json:"conditions,omitempty" patchStrategy:"
      merge" patchMergeKey:"type" protobuf:"bytes,1,rep,name=conditions"`
}

//+kubebuilder:object:root=true
//+kubebuilder:subresource:status
```

In this case, the standardized `Conditions` slice from the `metav1.Condition` struct is used to define the status of the custom resource. This is a common practice in Kubernetes, and it is used to define the status of the custom resource.

The `kokotap_controller.go` file will contain the logic to create the `kokotap_pod` pod, and it will be triggered by changes in the custom resource.

The control loop and how the newly kubernetes resource called `Kokotap` is broken down into pieces for better understanding and is shown below:

1. `func GetPodInfo:` This method fetches the pod information given the pod's name and namespace. It returns pod's `containerID`, and also the node's IP address (`nodeIP`) and node's name `nodeName` on which the pod is running, as well as the container runtime interface CRI being used in the cluster. All this information is needed afterwards to create `kokotap` resource.
2. `func CreateKokotapPod:` This method creates the `kokotap_pod` pod, and it is triggered by changes in the custom resource. It uses the `kokotap` container image, and it sets the necessary arguments to create the pod. The pod is created in the same node as the target pod, and it is used to mirror the traffic from the target pod, build the `VxLAN` tunnel and mirror the traffic to the external endpoint.
3. `func Reconcile:` This method reconciles the custom resource, it checks if the declared `Kokotap` resource needs to be created, deleted or updated and it calls other methods accordingly.
4. `func ReconcileNormal`: This method takes care of the creation of the pod that will execute the `kokotap_pod` binary. It creates the pod, and it sets the necessary arguments, specified in the `kokotap` resource `Specs`. It also sets the `finalizer` to the `Kokotap` resource, so the pod is not deleted until the `finalizer` is removed.
5. `func ReconcileDelete`: Handles the deletion state of the Kokotap resource, removing the `finalizer` and then the pod.
6. `func SetupWithManager:` This method sets up the custom controller with the manager, and it watches for changes in the custom resource. It is the main method that sets up the custom controller, and it is called by the main function.

The code snippets are shown below. First a snippet of `Reconcile` method:

Listing 4.12: Reconcile Method in controller.go file

```
func (r *KokotapReconciler) Reconcile(ctx context.Context, req ctrl.Request) (
    result ctrl.Result, errResult error) {
  logger := log.FromContext(ctx)
  logger.Info("Reconciling Kokotap")

  // Fetch the Kokotap instance
  kokotap := &networkingv1alpha1.Kokotap{}
  if err := r.Get(ctx, req.NamespacedName, kokotap); err != nil {
```

```
    if apierrors.IsNotFound(err) {
      logger.Info("Kokotap resource not found. Ignoring since object must be
          deleted")
      return reconcile.Result{}, nil
    }
    logger.Error(err, "Failed to get Kokotap object")
    return ctrl.Result{}, err
  }

  // Check if the Kokotap instance is marked to be deleted

  if kokotap.GetDeletionTimestamp() != nil {
    if controllerutil.ContainsFinalizer(kokotap, FinalizerName) {
      logger.Info("Kokotap Marked for deletion, deleting")
      return r.ReconcileDelete(ctx, kokotap)
    }
  }

  // Reconcile the Kokotap instance

  return r.ReconcileNormal(ctx, kokotap)

}
```

Then a short snippet of the `ReconcileNormal` method:

Listing 4.13: Reconcile Normal Method

```
// ReconcileNormal is the function that will be called when the resource is
    not being deleted or updated
func (r *KokotapReconciler) ReconcileNormal(ctx context.Context, kokotap *
    networkingv1alpha1.Kokotap) (ctrl.Result, error) {
  logger := log.FromContext(ctx)
  kokotapPod := &corev1.Pod{}
  err := r.Get(ctx, types.NamespacedName{Name: "kokotapped-" + kokotap.Spec.
      PodName, Namespace: kokotap.Spec.Namespace}, kokotapPod)
  if err != nil {
    if apierrors.IsNotFound(err) {
      logger.Info("Kokotap Pod not found. Creating one")
      _, err = r.CreateKokotapPod(ctx, kokotap)
      if err != nil {
        logger.Error(err, "Failed to create Kokotap Pod")
        return ctrl.Result{}, err
      }
      logger.Info("Created Kokotap Pod successfully")
      // Add finalizer to the Kokotap Custom Resource becasue the Kokotap Pod
          has been created
      if !controllerutil.ContainsFinalizer(kokotap, FinalizerName) {
        controllerutil.AddFinalizer(kokotap, FinalizerName)
        err = r.Update(ctx, kokotap)
        if err != nil {
          logger.Error(err, "Failed to add finalizer to Kokotap")
          return ctrl.Result{}, err
        }
        logger.Info("Added finalizer to Kokotap")
      }
      return ctrl.Result{}, nil
```

```
    }
    // Todo: Handle other errors
    logger.Error(err, "Failed to fetch Kokotap Pod")
    return ctrl.Result{}, err
  }
  return ctrl.Result{}, nil
}
```

And finally the `ReconcileDelete` method:

Listing 4.14: Reconcile Delete Method

```
// ReconcileDelete deletes the kokotap_pod and removes the label from the
    tapped pod
func (r *KokotapReconciler) ReconcileDelete(ctx context.Context, kokotap *
    networkingv1alpha1.Kokotap) (ctrl.Result, error) {

logger := log.FromContext(ctx)

// Delete the kokotap_pod and remove label from the tapped pod

podname := "kokotapped-" + kokotap.Spec.PodName
pod := &corev1.Pod{}
err := r.Get(ctx, types.NamespacedName{Name: podname, Namespace: kokotap.Spec.
    Namespace}, pod)
if err != nil {
    logger.Error(err, "Failed to get Pod")
    return ctrl.Result{}, err
  }

// Check kokotap status to handle deletion
if pod.Status.Phase == corev1.PodRunning {
    if err := r.Delete(ctx, pod); err != nil {
        logger.Error(err, "Failed to delete Pod")
        return ctrl.Result{}, err
      }
    logger.Info("Deleted Pod successfully")
  }

// Remove the label from the tapped pod
tappedpod := &corev1.Pod{}
err = r.Get(ctx, types.NamespacedName{Name: kokotap.Spec.PodName, Namespace:
    kokotap.Spec.Namespace}, tappedpod)
if err != nil {
    logger.Error(err, "Failed to get Pod")
    return ctrl.Result{}, err
  }
tappedpod.Labels["dn-vtap"] = "not-tapped"
if err := r.Update(ctx, tappedpod); err != nil {
    logger.Error(err, "Failed to update Pod")
    return ctrl.Result{}, err
  }

// Remove the finalizer from the Kokotap CR

controllerutil.RemoveFinalizer(kokotap, FinalizerName)
```

```
if err := r.Update(ctx, kokotap); err != nil {
    logger.Error(err, "Failed to remove finalizer from Kokotap")
    return ctrl.Result{}, err
  }
logger.Info("Removed finalizer from Kokotap")

return ctrl.Result{}, nil
}
```

The whole code repository will be attached in the appendix, and the user can check the full code in the repository. The code is also available in the following link: https://github.com/netand593/dn-vtap

### Deploying the Custom Resource Definition

After defining the custom resource and the custom controller, the user can deploy the custom resource definition to the Kubernetes cluster. Following the Operator SDK documentation. This will, once again, scaffold all the manifests that the controller needs, to be able to manage the new kubernetes resource. Some important files that are generated are:

1. `config/crd/bases/networking.dn-lab.io_kokotaps.yaml`: This file contains the definition of the custom resource, and it must be deployed to the Kubernetes cluster with a simple `kubectl apply -f` command.
2. `config/samples/networking_v1alpha1_kokotap.yaml`: This file contains a sample of the custom resource, and it can be used to test the custom resource in the Kubernetes cluster.
3. `config/rbac`: This directory contain multiple files to create Kubernetes security objects according to the CRD and best practices. To name some objects it creates a `ClusterRole`, a `ClusterRoleBinding` and a `ServiceAccount` for the operator, so it is actually authorized to handle the newly created resource.

The user can deploy the custom resource definition to the Kubernetes cluster with the following command:

Listing 4.15: Deploying the Custom Resource Definition

```
$ kubectl apply -f config/crd/bases/networking.dn-lab.io_kokotaps.yaml
customresourcedefinition.apiextensions.k8s.io/kokotaps.networking.dn-lab.io
    created
```

The user can then create a new custom resource with the following command:

Listing 4.16: Creating a new Custom Resource

```
$ kubectl apply -f networking_v1alpha1_kokotap.yaml
kokotap.networking.dn-lab.io/kokotap-sample created
```

A sample of how the YAML file for the custom resource looks like is shown below:

Listing 4.17: Sample of the Kokotap CRD

```
apiVersion: networking.dn-lab.io/v1alpha1
kind: Kokotap
metadata:
  labels:
    app.kubernetes.io/name: kokotap
    app.kubernetes.io/instance: kokotap-sample
    app.kubernetes.io/part-of: dn-vtap
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: dn-vtap
  name: kokotap-test-1
spec:
  podName: "ueransim-gnb-57674b4b94-pv5rn"
  destIp: "192.168.1.109"
  vxlanID: 1100
  namespace: "default"
  mirrorType: "both"
  podInterface: "eth0"
  image: "netand593/kokotap:2.0-beta"
```

The user can then check the status of the custom resource with the following command:

Listing 4.18: Checking the status of the Custom Resource

```
$ kubectl get kokotap
NAME              AGE
kokotap-test-1    6m30s
```

All the commands for a common, predefined kubernetes resource could be use here, like `kubectl describe`, `kubectl edit`, `kubectl delete` and so on.

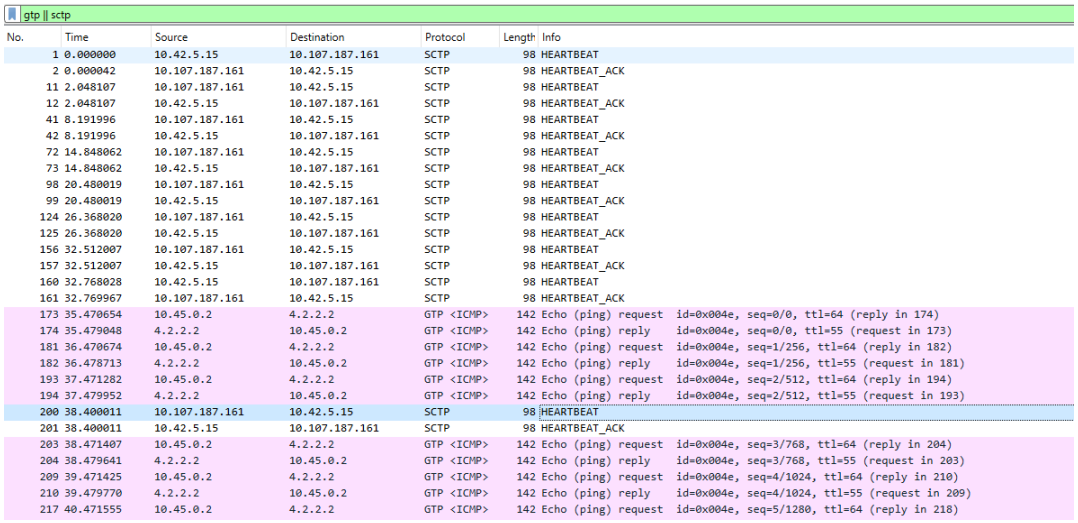
Similarly as explained in section 4.1.1, the custom controller will in this case create the kokotap `capturing` or `mirroring` pod, and the user can then check the status of the pod with the following command:

Listing 4.19: Checking the status of the Kokotap Pod

```
$ sudo tcpdump -vvvi vxlan0 -s 65535 -w kokotap-crd-test-1-gnb.pcap
[sudo] password for andres:
tcpdump: listening on vxlan0, link-type EN10MB (Ethernet), capture size 65535
    bytes
^C323 packets captured
325 packets received by filter
0 packets dropped by kernel
```

Some test traffic was generated as well from the UE pod, and this traffic is being intercepted in the gNB pod, and then sent to the external endpoint. The user can then analyze the captured traffic using Wireshark, and the results should be similar to the ones shown in the figure below. For the sake of proving the test, a filter was applied to only show GTP traffic, which is the protocol used in 5G networks to encapsulate user data.

The capture is shown in the figure below:



gtp || sctp

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 10.42.5.15 | 10.107.187.161 | SCTP | 98 | HEARTBEAT |
| 2 | 0.000042 | 10.107.187.161 | 10.42.5.15 | SCTP | 98 | HEARTBEAT_ACK |
| 11 | 2.048107 | 10.107.187.161 | 10.42.5.15 | SCTP | 98 | HEARTBEAT |
| 12 | 2.048107 | 10.42.5.15 | 10.107.187.161 | SCTP | 98 | HEARTBEAT_ACK |
| 41 | 8.191996 | 10.107.187.161 | 10.42.5.15 | SCTP | 98 | HEARTBEAT |
| 42 | 8.191996 | 10.42.5.15 | 10.107.187.161 | SCTP | 98 | HEARTBEAT_ACK |
| 72 | 14.848062 | 10.107.187.161 | 10.42.5.15 | SCTP | 98 | HEARTBEAT |
| 73 | 14.848062 | 10.42.5.15 | 10.107.187.161 | SCTP | 98 | HEARTBEAT_ACK |
| 98 | 20.480019 | 10.107.187.161 | 10.42.5.15 | SCTP | 98 | HEARTBEAT |
| 99 | 20.480019 | 10.42.5.15 | 10.107.187.161 | SCTP | 98 | HEARTBEAT_ACK |
| 124 | 26.368020 | 10.107.187.161 | 10.42.5.15 | SCTP | 98 | HEARTBEAT |
| 125 | 26.368020 | 10.42.5.15 | 10.107.187.161 | SCTP | 98 | HEARTBEAT_ACK |
| 156 | 32.512007 | 10.107.187.161 | 10.42.5.15 | SCTP | 98 | HEARTBEAT |
| 157 | 32.512007 | 10.42.5.15 | 10.107.187.161 | SCTP | 98 | HEARTBEAT_ACK |
| 160 | 32.768028 | 10.42.5.15 | 10.107.187.161 | SCTP | 98 | HEARTBEAT |
| 161 | 32.769967 | 10.107.187.161 | 10.42.5.15 | SCTP | 98 | HEARTBEAT_ACK |
| 173 | 35.470654 | 10.45.0.2 | 4.2.2.2 | GTP <ICMP> | 142 | Echo (ping) request id=0x004e, seq=0/0, ttl=64 (reply in 174) |
| 174 | 35.479048 | 4.2.2.2 | 10.45.0.2 | GTP <ICMP> | 142 | Echo (ping) reply id=0x004e, seq=0/0, ttl=55 (request in 173) |
| 181 | 36.470674 | 10.45.0.2 | 4.2.2.2 | GTP <ICMP> | 142 | Echo (ping) request id=0x004e, seq=1/256, ttl=64 (reply in 182) |
| 182 | 36.478713 | 4.2.2.2 | 10.45.0.2 | GTP <ICMP> | 142 | Echo (ping) reply id=0x004e, seq=1/256, ttl=55 (request in 181) |
| 193 | 37.471282 | 10.45.0.2 | 4.2.2.2 | GTP <ICMP> | 142 | Echo (ping) request id=0x004e, seq=2/512, ttl=64 (reply in 194) |
| 194 | 37.479952 | 4.2.2.2 | 10.45.0.2 | GTP <ICMP> | 142 | Echo (ping) reply id=0x004e, seq=2/512, ttl=55 (request in 193) |
| 200 | 38.400011 | 10.107.187.161 | 10.42.5.15 | SCTP | 98 | HEARTBEAT |
| 201 | 38.400011 | 10.42.5.15 | 10.107.187.161 | SCTP | 98 | HEARTBEAT_ACK |
| 203 | 38.471407 | 10.45.0.2 | 4.2.2.2 | GTP <ICMP> | 142 | Echo (ping) request id=0x004e, seq=3/768, ttl=64 (reply in 204) |
| 204 | 38.479641 | 4.2.2.2 | 10.45.0.2 | GTP <ICMP> | 142 | Echo (ping) reply id=0x004e, seq=3/768, ttl=55 (request in 203) |
| 209 | 39.471425 | 10.45.0.2 | 4.2.2.2 | GTP <ICMP> | 142 | Echo (ping) request id=0x004e, seq=4/1024, ttl=64 (reply in 210) |
| 210 | 39.479770 | 4.2.2.2 | 10.45.0.2 | GTP <ICMP> | 142 | Echo (ping) reply id=0x004e, seq=4/1024, ttl=55 (request in 209) |
| 217 | 40.471555 | 10.45.0.2 | 4.2.2.2 | GTP <ICMP> | 142 | Echo (ping) request id=0x004e, seq=5/1280, ttl=64 (reply in 218) |

Figure 4.4: Wireshark capture of the traffic, filtered to show only GTP. Captured using Kokotap CRD.

The test as shown was successful, the objective of sending "live" traffic to an external endpoint was duly accomplished.

## 4.3 Istio Service Mesh with Kiali

As seen in section 2.5.3 the Istio service mesh is a powerful tool to manage and monitor the traffic in a Kubernetes cluster. Istio provides a lot of features to manage the traffic, and it is a very powerful tool to monitor the traffic in a Kubernetes cluster. One of the most powerful tools that Istio provides is Kiali, which is a graphical user interface to monitor the traffic in a Kubernetes cluster.

### 4.3.1 Motivation for Istio and Kiali

Istio provides an additional plugin for monitoring the traffic in a Kubernetes cluster, and it is called Kiali. Kiali is an observability console for Istio service mesh, providing detailed insights into the service mesh components. [11] It helps in visualizing the service mesh topology, understanding the health of the services, and managing traffic between the services. Kiali integrates seamlessly with Istio, leveraging its capabilities to monitor and manage microservices. Kiali takes advantage of Istio's *sidecar injection* feature, which injects a sidecar container into each pod, and it provides a lot of features to monitor the traffic in a Kubernetes cluster. The most important features that can contribute to fulfill objectives are:

- **Service Graph**: Kiali provides a service graph that shows the topology of the services in the Kubernetes cluster. The service graph shows the services of the application and how they are functionally connected to each other, meaning how they are exchanging traffic.
- **Connect tracing services:** Kiali provides seamless integration with *tracing* services like *Jaeger*, which is a distributed tracing system, and it is used to monitor and troubleshoot microservices-based distributed systems. Jaeger could serve as a way to keep track of flows of traffic across the cluster and to identify anomalous behavior.

### 4.3.2 Deploying Istio and Kiali

The deployment of Istio and Kiali is a complex process, as it requires first to install and configure the Service Mesh from Istio and then deploy a custom controller for Kiali, that integrates and sets up all the functionalities for Traffic Monitoring and the integration with *Jaeger* distributed tracing services. The user can follow the official documentation to deploy Istio and Kiali in a Kubernetes cluster which provides enough information to carry out the process.[2]

Istio is easily installed with the proper `helm` chart, it can be accomplished as follow:

Listing 4.20: Installing Istio with Helm

```
andres@k8s-cp:~/kiali$ helm install istio-base istio/base -f istio-change.yaml
     --reuse-values -n istio-system
Release "istio-base" has been upgraded. Happy Helming!
NAME: istio-base
LAST DEPLOYED: Tue Jan 23 22:56:15 2024
NAMESPACE: istio-system
STATUS: deployed
```

[2] https://istio.io/latest/docs/setup/getting-started/

```
REVISION: 3
TEST SUITE: None
NOTES:
Istio base successfully installed!
```

And the Custom Controller, which is a very important part of the deployment is installed by applying the following yaml file:

Listing 4.21: Installing Kiali Custom Controller

```
apiVersion: kiali.io/v1alpha1
kind: Kiali
metadata:
  name: kiali
  namespace: istio-system
spec:
  auth:
    strategy: "anonymous"
  deployment:
    accessible_namespaces: ["bookinfo", "open5gs"]
    view_only_mode: false
    service_type: "LoadBalancer"
  server:
    web_root: "/kiali"
  external_services:
    tracing:
      # Enabled by default. Kiali will anyway fallback to disabled if
      # Jaeger is unreachable.
      enabled: true
      provider: "jaeger"
      # if you set "use_grpc" to false.
      in_cluster_url: "http://jaeger-query.istio-system:16686"
      use_grpc: true
      auth:
        ca_file: ""
        insecure_skip_verify: false
        password: ""
        token: ""
        type: "none"
        use_kiali_token: false
        username: ""
---
```

After several test the user can get a visual representation of the traffic in the Kubernetes cluster, and the user can see the services that are exchanging traffic, and how they are connected to each other. Please refer to the following figure to see how the Kiali dashboard looks like:

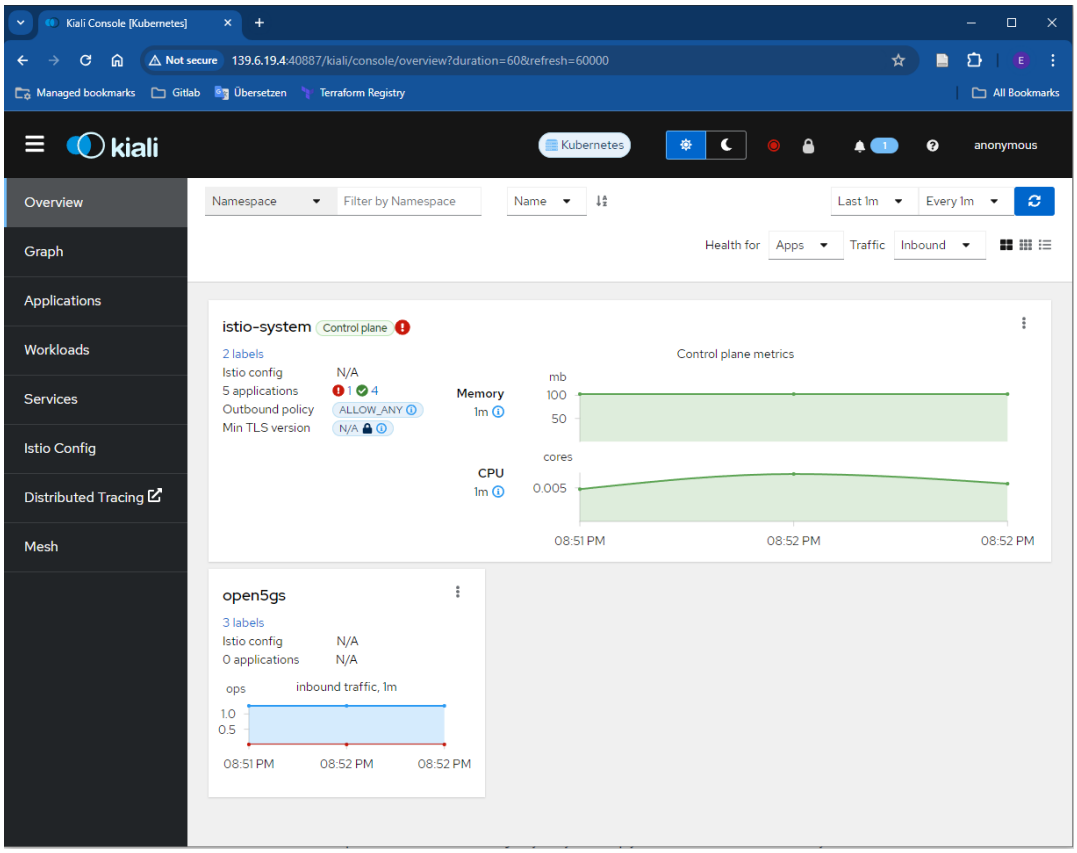



Figure 4.5: Kiali Dashboard showing the traffic in the Kubernetes cluster

The user can see the services that are exchanging traffic, and how they are connected to each other, by using the graphic view:

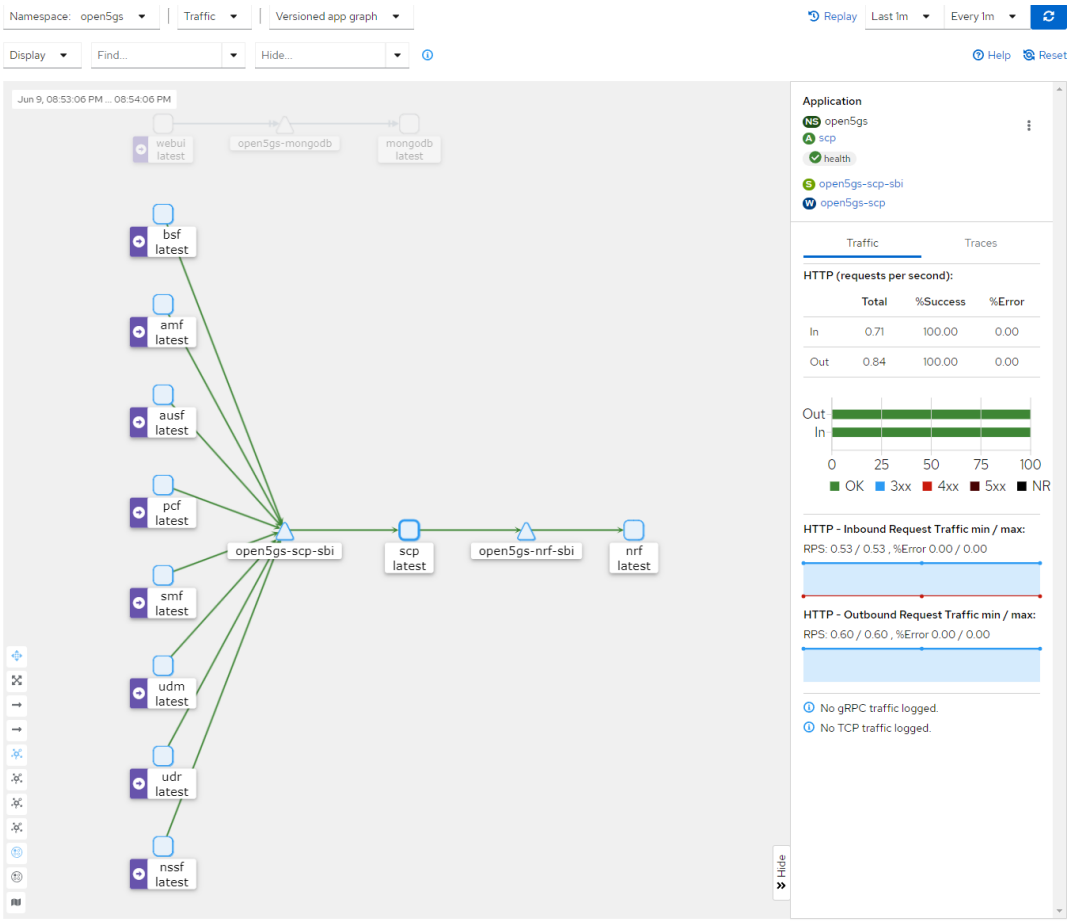



Figure 4.6: Kiali Graph showing the services in the Kubernetes cluster

By clicking on one individual *Kubernetes Service*, the administrator can take a look into the several important metrics, like Inbound and Outbound traffic, logs received from the pod associated with the service, like shown below:

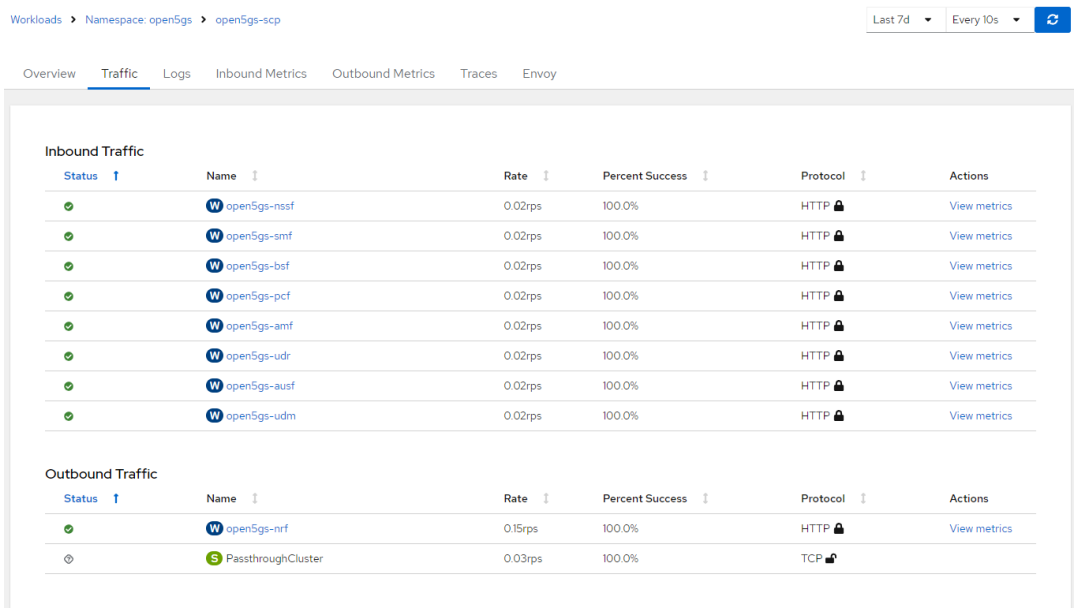



Figure 4.7: Kiali Service View showing the metrics of the service

If we refer to Figure 4.5, in the left side menu there is a button to access the *Tracing* view, which is a very powerful tool to monitor the traffic in the Kubernetes cluster. This leads to the Jaeger dashboard, which shows all the traces of the traffic in the Kubernetes cluster, in a per-service fashion, as shown in the figure below:

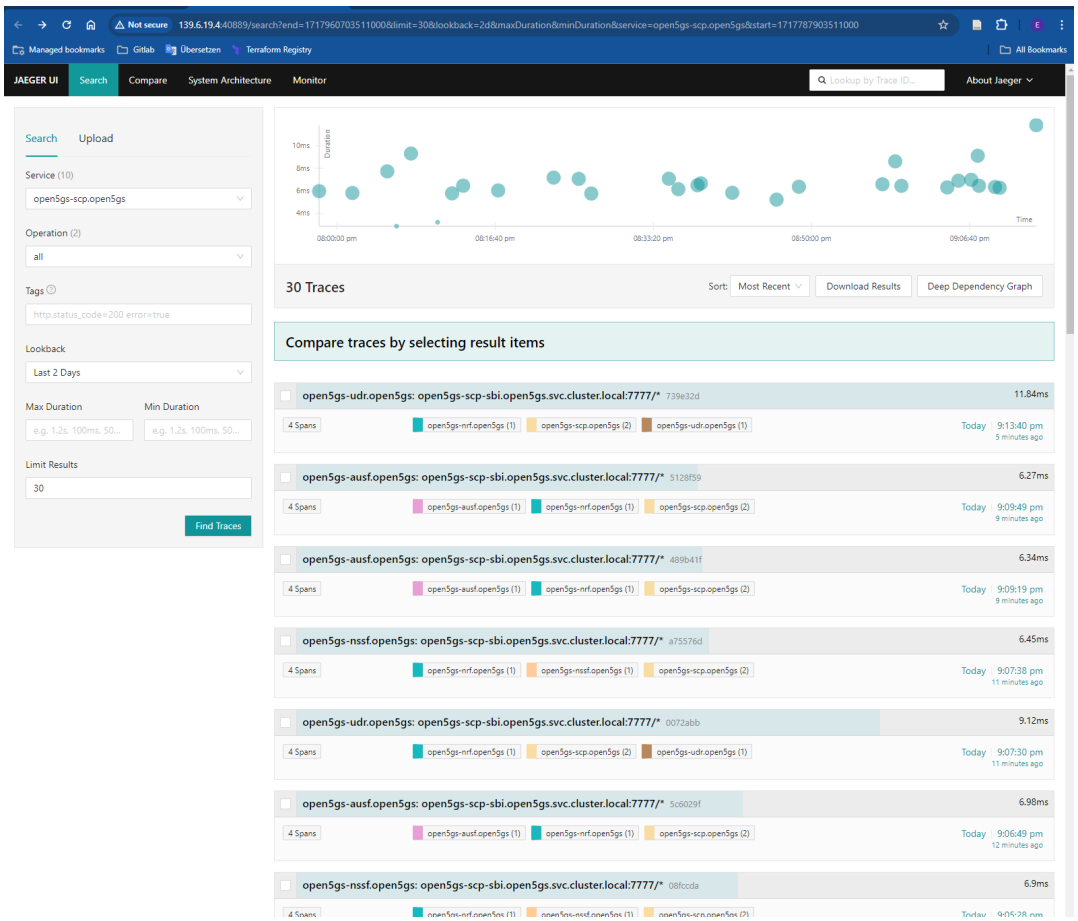



Figure 4.8: Jaeger Tracing showing the traces of the traffic in the Kubernetes cluster

## 4.4 Comparison of the Strategies

The three strategies presented in this chapter are valid tools that accomplish the objective of the project, and provide flexibility for future works to choose any of them, depending on the approach that the developer or the user in general wants to incorporate in future works. A comparison of the three strategies is shown below:

| Strategy | Pros | Cons | Use Case |
|---|---|---|---|
| **Kokotap CLI** | Easy to use<br>Rapid deployment<br>Single binary that executes from the CP node | Not K8s managed<br>Ephemeral nature (not resilient to changes)<br>Limited CRI support<br>Very few development | Quick tests<br>Dev clusters |
| **Kokotap CRD** | In-band management (kubectl commands)<br>Non-ephemeral behavior<br>Flexible and multiple deployments | Relies on Kokotap development (very poor)<br>Initial complex development<br>Security needs to be refined | Prod and dev clusters |
| **Istio Service Mesh** | Consolidated product<br>Broader support by companies and CNCF<br>Rich set of features | Complex setup<br>Sidecar injection adds complexity and management workload<br>Some paid features<br>SCTP support not provided | Production clusters<br>Highly scalable clusters |

Table 4.1: Comparison of the Strategies

### 4.4.1 Evaluation of Strategies

The virtual TAP and monitoring strategies are evaluated based on the following criteria:

- **Ease of Use**: The ease of use of the strategy, and how easy it is to deploy and manage the strategy.
- **Deployment Time**: The time it takes to deploy the strategy, and how fast it is to deploy the strategy.
- **Development**: The development of the strategy, and how easy it is to develop the strategy.
- **5G compatibility**: Meaning how well does the strategy adapts to the 5G setup.

Regarding the easiness of use, the Kokotap CLI strategy is the easiest to use, as it is a single binary that can be executed from the CP node. The Kokotap CRD strategy

is also easy to use, as it is managed by Kubernetes, and it can be deployed with `kubectl` commands. The Istio Service Mesh strategy is the most complex to use, as it requires a lot of setup and configuration, as well as much more flexible communication between services inside the cluster, therefore influencing the initial design of the 5G core deployment or maybe bring additional workload to deployments which are already in production.

In the matter of deployment time, again Kokotap CLI is the quickest way, as it only requires to download the binaries from `Github` and execute it as expected. The Kokotap CRD on the other hand, required more effort as it is a newly designed solution, but the use and deployment was very quick once the tests were successful, as it is just another kubernetes resource which is administrated via `kubectl` commands. The Istio Service Mesh with Kiali and Jaeger plugins, although it is more complex, didn't require much time to deploy, since it is a very well documented product and the community around it provide excellent tutorials and quick support for any questions.

In the development aspect, the Kokotap CLI strategy is the most limited, as it has been abandoned by its developers and the community has not showed further interest in keeping it alive and available for production clusters, and it is not managed by Kubernetes. The Kokotap CRD arises a development concern as well, since it is a brand new development for this particular thesis and also relies on the `kokotap` binaries and container images, although could be pushed forward by future works. The Istio Service Mesh strategy relies on the constant updates and feature releases that the `Istio` community and development team provide, and it is a very well maintained product, with a lot of features and a lot of support from the community. It is also offered as a managed solutions in some public cloud solutions like `Google Cloud Platform` and `IBM Cloud`.

In the 5G compatibility aspect, being one of the most important for the project; both `Kokotap` CLI and CRD fit the 5G setup, as they are agnostic to the type of traffic to be analyzed, while on the other hand `Istio` with `Kiali` and `Jaeger` pose a huge limitation for 5G traffic, as it only cares about much more common traffic to be found in Kubernetes applications as HTTP, gRPC and TCP traffic.

# 5 Conclusions and Future Work

As a brief summary, in this thesis the evaluation of virtual TAP and monitoring alternatives for 5G virtualized environments was carried out by following the following very general steps:

1. **Literature Review:** A comprehensive literature review was conducted to understand the state-of-the-art in 5G networks, virtualization, monitoring, and TAP technologies.
2. **Requirements Analysis:** The requirements for a 5G monitoring solution were identified and documented.
3. **Design and Implementation:** A testbed was designed and implemented to evaluate the performance of virtual TAP and monitoring alternatives for 5G networks.
4. **Proposed solutions:** Three main solutions were proposed to address the challenges of virtual TAP and monitoring in 5G networks.
5. **Evaluation:** The performance of the proposed solutions was evaluated in terms of throughput, latency, and scalability.

After the evaluation of the proposed solutions, the following conclusions can be drawn:

## 5.1 Conclusions

The results of the evaluation showed that the proposed solutions can provide a viable alternative to traditional TAP and monitoring solutions for 5G networks. The proposed solutions were able to achieve high throughput, low latency, and good scalability, which are essential for monitoring high-speed 5G networks. The proposed solutions can be easily integrated into existing 5G networks and can be used to monitor the performance of 5G networks in real-time.

The contributions of this thesis can be summarized as follows:

- Since there is a very strong trend to move all possible applications to the cloud, the mobile networks as a service are also moving to the cloud. Most of the top-tier operators across the world are trying to accelerate their migration to public and private cloud environments, and locally in Germany are also heading to this transformation. Given the numerous set of alternatives to offer a fully functional 5G Network in the cloud, that follow a Microservices architecture, Kubernetes is the top-choice when deploying an application in the cloud, since it orchestrates such architectures in a very consistent way and encompass a very large set of tools to monitor and manage the network, all supported by a very active multidisciplinary community.

  Given the security concerns that a cloud deployment in general bring up, in conjunction with the delicate nature (security-wise) of mobile networks, it is an absolute necessity to provide a *traffic analysis* solution that adapts to such a scenario accordingly.

- The proposed solutions can provide a viable alternative to traditional TAP and monitoring solutions for 5G networks. The proposed solutions were able to achieve high throughput, low latency, and good scalability, which are essential for monitoring high-speed 5G networks. The proposed solutions can be easily integrated into existing 5G networks and can be used to monitor the performance of 5G networks in real-time.

- The Kokotap CLI solution is a very quick alternative to spin up a virtual tap in test clusters. It relies also on the feasibility of creating a VXLAN that connects the worker node on which the target pod is running and the external node or *collector* that will receive the traffic. However, the solution is not easily scalable, since it requires a manual intervention to create the VXLAN interfaces, and has been abandoned by its developer team. It wouldn't fit well in production environmets, given that multiple pods would require a manual intervention to create the VXLAN interfaces. Some adjustments to the code was actually necessary to make it compatible with the latest versions of Kubernetes.

- The monitoring alternatives provided by a service mesh like *Istio* are very powerful. It not only provides a very detailed view of the network, but also the alternative to incorporate multiple *tracing* like *Jaeger* and *Grafana Tempo* to collect and analyze the data. Istio and Kiali can offer a graphical view of the network, even with the type of traffic flowing through network functions within the 5G core, which can benefit network operators and specialist to better comprehension of the network.

  However, for this particular case of a 5G network, the Istio and Kiali do not offer monitor or traceability of the control plane traffic, since most of it consist

of SCTP and GTP packets, which are not yet supported by the service mesh. It must be considered too, that the service mesh itself is a complex solution, which requires additional computing resources to be used and comprehension of the concept of *sidecar injection*, and includes an additional layer of management workload for the cluster.

- The proposed solution of creating a new CRD is a very powerful alternative to create a virtual TAP in Kubernetes. It is very scalable, since it can be easily integrated with the Kubernetes API, and can be used to mirror traffic on a per-pod basis. The main benefit of this solution is that the user doesn't depend on a single binary executing in the control plane node of the cluster, it can be easily managed by any user with the right permissions to create a *custom resource* in the cluster. At the same time, a lightweight, written in go, kubernetes controller, makes it *in-band managed* by Kubernetes, following the cloud native approach and the Kubernetes philosophy (all managed through the kubernetes API). It still depends on the feature offered by Kokotap, since the *mirroring-pod* is still reliant on the kokotap code, but it can be easily extended as it is written in go too.

Summarizing the results of the evaluation, it can be concluded that the solution that best fits the requirements of a 5G network is the one that uses a CRD to create a virtual TAP in Kubernetes, and future work could even improve and lead to a more robust solution, that could be used in production environments.

## 5.2 Future Work

The proposed solutions can be further improved and extended in several ways. Some of the possible future work includes:

- **Support for control plane traffic:** The proposed solutions do not support monitoring of control plane traffic, which is an essential part of a 5G network. Future work could focus on extending the solutions to support monitoring of control plane traffic.
- **Extend kokotap code further:** Kokotap source code needs to be extended and maintained to be always compatible with new versions of kubernetes, maybe not major changes would occur soon, but some minor adjustments might break the functionality of both Kokotap CLI and CRD. Future work could focus on integrating all CRI in the market (now it only supports CRI-O and docker), better error logging and handling, and a more robust way to create the VXLAN interfaces. Even look into alternatives for traffic mirror than `tc-mirred`, a good

approach would be to create gRPC calls to the control plane node to mirror the traffic.

- **Investigate eBPF alternatives:** The use of eBPF to create a virtual TAP in Kubernetes is a very promising alternative. Future work could include the development of a custom controller that uses eBPF to define rules for mirroring traffic in Kubernetes.

  Since `eBPF` is a tool built on Linux kernel that allows the very powerful tool, it can be used to create a very efficient and scalable solution for monitoring 5G networks. The approach could go in two different ways: the first one, to work on adding SCTP support to `Cilium`, which is a CNI alternative that brings excellent observability and monitoring tools (among many others) that a 5G core could benefit from. The second one, to create a new eBPF program that could be used to mirror traffic in Kubernetes, and wrap it into a custom controller that can be easily deployed and managed via `kubectl` commands.

- **Research Kubernetes Gateway API option:** Gateway API is another native tool offered by Kubernetes, which establishes a new method to manage and route traffic from service to service within the cluster [13]. In this case a very interesting solution would be to create rules to mirror traffic from a pod to another gateway or the control plane node itself and analyze the traffic further. Another more robust solution would be a custom controller that manages the Gateway API resources and creates/deletes/modifies the capturing rules as needed.

- **Integration with 5G network functions:** The proposed solutions could be further extended to integrate with 5G network functions. Future work could focus on developing a custom controller that can be used to monitor the performance of 5G network functions in real-time. In a very ambitious approach this can be accomplished by adding this feature into service meshes solutions like Istio or Linkerd, which currently does not support the monitoring of control plane traffic (from 5G core, not to be confused with *control plane node* from a kubernetes cluster).

# Bibliography

The following sources were cited alongside the writting.


[1] Brendan Burns et al. *Kubernetes: up and running.* " O'Reilly Media, Inc.", 2022.

[2] Cilium. *Cilium SCTP Support (beta).* Online; accessed 2023-12-15. 2023. URL: https://docs.cilium.io/en/latest/configuration/sctp/.

[3] Matt Farina and Josh Dolitsky. *Learning Helm.* O'Reilly, 2022.

[4] Cloud-Native Computing Foundation. *Cluster Networking.* Online; accessed 2023-12-15. 2023. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[5] Free5GC. *Free5GC Documentation.* Online; accessed 2024-02-12. 2023. URL: https://www.free5gc.org/docs/.

[6] Evan Gilman and Doug Barth. *Zero trust networks.* O'Reilly Media, Incorporated, 2017.

[7] Michael Hausenblas and Stefan Schimanski. *Programming Kubernetes: Developing cloud-native applications.* O'Reilly Media, 2019.

[8] Gigamon Inc. *Direct cabling vs. TAP cabling.* Online; accessed 2023-12-12. 2023. URL: https://www.gigamon.com/content/dam/website-assets/network-diagrams/WP-Understanding-Network-TAPs-10.19_06-Diagram-1-600x314.jpg.imgo.jpg.

[9] Gigamon Inc. *Understanding Network TAPs.* Online; accessed 2023-12-14. 2023. URL: https://www.gigamon.com/products/access-traffic/network-taps.html.

[10] Google Inc. *Istio Service Mesh.* Online; accessed 2024-03-07. 2022. URL: https://cloud.google.com/learn/what-is-istio.

[11] Alice Jones and Bob Smith. "Visualizing and Managing Microservices with Kiali". In: *Journal of Microservices* 12.3 (2021). Accessed: 2024-06-09, pp. 45–56. URL: https://journals.microservices.com/kiali-2021.

[12] Kubernetes. *Cluster Architecture.* Online; accessed 2024-03-15. 2022. URL: https://kubernetes.io/docs/concepts/architecture/.

[13] Kubernetes. *Gateway API*. Online; accessed 2023-12-15. 2023. URL: https://kubernetes.io/docs/concepts/services-networking/gateway/.

[14] Open5GS. *Open5GS Documentation*. Online; accessed 2024-02-12. 2023. URL: https://open5gs.org/open5gs/docs/.

[15] Chris Richardson. *Microservice Architecture pattern*. Online; accessed 2023-08-12. 2023. URL: https://microservices.io/patterns/microservices.html.

[16] Amazon Web Services. *What is a Service Mesh?* Online; accessed 2024-03-07. 2023. URL: https://aws.amazon.com/what-is/service-mesh/?nc1=h_ls.

[17] James Strong and Vallery Lancey. *Networking and Kubernetes*. " O'Reilly Media, Inc.", 2021.

[18] Paul Sutton and srsRAN comitte. *srsRAN Project*. Online; accessed 2024-02-10. 2024. URL: https://github.com/srsran/srsRAN_Project.

[19] ueransim. *ueransim GitHub Repository*. Online; accessed 2024-02-13. 2023. URL: https://github.com/aligungr/UERANSIM.

# Appendix

## .1 Appendix A

### .1.1 Tables

## Comparison of Kubernetes, Mesos, and Nomad as Container Orchestration Tools

Table .1: Comparison of Kubernetes, Mesos, and Nomad as Container Orchestration Tools

| **Feature** | **Kubernetes** | **Mesos (Apache Mesos)** | **Nomad** |
|---|---|---|---|
| **Architecture** | Master-worker architecture with high availability. Complex setup but robust in handling large clusters. | Master-agent architecture that can run other frameworks such as Marathon. Suited for very large and heterogeneous clusters. | Single binary that can run as both server and client. Simple architecture ideal for both small and large deployments. |
| **Scalability** | Designed to handle up to thousands of nodes seamlessly. Auto-scaling capabilities are built-in. | Excellently scales up to tens of thousands of nodes. Known for managing massive scale deployments. | Scales well and is efficient in resource usage, though generally considered behind Kubernetes in maximum capacity. |
| **Usability** | Complex setup with a steep learning curve. Rich feature set that requires deeper understanding. | Complex because it can run multiple types of workloads and can integrate with other frameworks. | Simple and straightforward to setup and use, with less complexity in operations. |
| **Ecosystem** | Extensive ecosystem with a wide range of tools and integrations developed by a large community and significant corporate backing. | Good ecosystem with support for many cluster-level applications and frameworks but less vibrant compared to Kubernetes. | Smaller ecosystem but growing. Focuses on simplicity and integrability. |
| **Workload Diversity** | Supports a wide variety of workloads, including stateless, stateful, and data-processing workloads. | Originally designed for data-heavy and compute-intensive applications but supports a variety of tasks via frameworks. | Primarily focused on containerized and non-containerized applications. Less native support for complex data processing unless integrated with other tools. |
| **License** | Open-source under the Apache License 2.0. | Open-source under the Apache License 2.0. | Open-source under the Mozilla Public License 2.0. |

**Comparison of CNIs for Kubernetes: Calico, Cilium, and Flannel**

Table .2: Comparison of CNIs for Kubernetes: Calico, Cilium, and Flannel

| **Feature** | **Calico** | **Cilium** | **Flannel** |
|---|---|---|---|
| **Network Model** | Layer 3 based networking that supports advanced routing options. No overlay by default, supports BGP. | Layer 3/4 networking that supports advanced routing and direct connectivity with BPF and XDP. | Simple overlay network that uses either VXLAN or UDP to encapsulate IPIP packets. |
| **Network Policies** | Supports standard Kubernetes network policies and extends them with Calico-specific enhancements. | Offers extensive network security options, including identity-based security policies via BPF. | Basic support for Kubernetes network policies but lacks advanced features found in Calico or Cilium. |
| **Performance** | Very high performance in native networking environments, potentially reduced in overlay configurations. | Extremely high performance, especially in environments that leverage BPF for data plane operations. | Generally good performance, best suited for smaller or simpler deployments. |
| **Ease of Use** | Moderate complexity due to advanced feature set. Steeper learning curve for full feature utilization. | Complex setup due to advanced capabilities and dependencies on newer kernel features. | Easy to set up and use, with minimal configuration required. Ideal for beginners or simple use cases. |
| **Integration** | Deep integration options with standard Kubernetes environments and cloud-native ecosystems. | Advanced integrations, particularly in security-focused environments, with strong ties to cloud-native security tools. | Limited integration capabilities compared to Calico and Cilium. |
| **Use Case** | Suited for large-scale, security-sensitive, or performance-intensive deployments. | Ideal for security and performance-intensive applications, especially those requiring deep packet inspection. | Best for smaller, less complex networks or for users just starting with Kubernetes. |

**Comparison of Container Runtimes for Kubernetes: Docker, cri-o, and containerd**

Table .3: Comparison of Container Runtimes for Kubernetes: Docker, cri-o, and containerd

| **Feature** | **Docker** | **cri-o** | **containerd** |
|---|---|---|---|
| **Kubernetes Compatibility** | Initially the default in Kubernetes; now Kubernetes has deprecated Docker in favor of runtimes that use the Container Runtime Interface (CRI). | Designed specifically to integrate with Kubernetes, fully compatible as a direct CRI. | Also uses CRI, making it fully compatible with Kubernetes. Derived from components of Docker. |
| **Performance** | Good performance, but can be heavier due to additional features not required by Kubernetes. | Lightweight and optimized for Kubernetes, potentially offering better performance, developed by RedHat. | Lightweight and efficient, similar to cri-o, as it focuses only on core container runtime tasks. |
| **Security Features** | Provides comprehensive security features, but some are dependent on enterprise editions. | Minimal attack surface due to a narrower scope of functionality, enhancing security. | Shares similar security benefits with cri-o by focusing on simplicity and minimalism. |
| **Community Support** | Very large community support, extensive documentation, and a wide ecosystem but deprecated. | Growing community, with support primarily driven by Kubernetes users and Red Hat. | Strong community support, governed by the Cloud Native Computing Foundation (CNCF) along with Kubernetes. |
| **Use Case** | Suited for development environments and smaller production environments not strictly using Kubernetes. | Best for those fully invested in Kubernetes, needing a tailored and optimized runtime. | Ideal for users who need a simple, robust, and Kubernetes-focused container runtime without additional overhead. |

**Comparison of Open5GS and Free5GC for 5G Core Network Implementation**

Table .4: Comparison of Open5GS and Free5GC for 5G Core Network Implementation

| **Feature** | **Open5GS** | **Free5GC** |
|---|---|---|
| **Architecture** | Implements both 4G (EPC) and 5G (5GC) core network functionalities. Modular design with individual network functions (NFs) that can be deployed independently. | Focused primarily on 5G core network functions. Implements a comprehensive set of 5G NFs, adhering closely to 3GPP standards. |
| **Ease of Use** | User-friendly with detailed documentation and guides. Suitable for both educational and production environments. | More complex to set up due to its focus on 5G, but extensive documentation is available. Better suited for research and advanced testing. |
| **Performance** | Optimized for both development and production use, with a balance between simplicity and performance. | High performance in 5G scenarios, often used for research projects that require detailed protocol adherence and testing. |
| **Network Functions (NFs)** | Supports a wide range of NFs including allowing 5G Stand-Alone and Non Standalone deployments. | Comprehensive support for 5G Standalone-compliant NFs such as AMF, SMF, UPF, AUSF, UDM, PCF, NSSF, NRF, etc. |
| **Community and Support** | Active community with extensive support through GitHub, mailing lists, and forums. Frequently updated with new features and fixes. | Strong community presence, particularly in academic and research circles. Support is available through GitHub and other online resources. |
| **Deployment** | Can be deployed on various platforms including Docker, Kubernetes, and bare-metal servers. Detailed Helm charts available for Kubernetes deployments. | Primarily targeted at research and testing environments, with support for Docker and Kubernetes deployments. Helm charts and scripts are available for easier deployment. |
| **Use Case** | Ideal for both educational purposes and real-world production deployments. Flexible and modular, making it suitable for various scales of deployment. | Best suited for research, development, and testing of 5G networks. Highly detailed implementation for in-depth protocol study and experimentation. |

# Declaration

I declare that I have written this thesis independently. I have labelled all passages that are taken verbatim or in spirit from published or unpublished works by others or by the author him/herself as having been taken. All sources and aids that I have used for the work are indicated. The work has not yet been submitted to any other examination authority with the same content or in essential parts.

Cologne, 10.06.2024

Location, Date

Signature