ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

DIPARTMENT OF INFORMATION – SCIENCE AND ENGINEERING

(DISI)

Master's Degree in Computer Science Engineering

Thesis

in

Distributed Systems M

Deployment and Orchestration in Cloud Native 5G Networks with Kubernetes

Candidate:

Asma Noor

Supervisor:

Chiar.mo Prof. Paolo Bellavista

Co-Supervisor:

Chiar.mo Prof. Luca Foschini

Prof. Ing. Domenico Scotece

Academic Year 2021/22

Table of Contents

Abstract

Software Defined Networking along with Network Function Virtualisation have brought an evolution in the telecommunications laying out the bases for 5G networks and its softwarisation. The separation between the data plane and the control plane, along with having a decentralisation of the latter, have allowed to have a better scalability and reliability while reducing the latency. A lot of effort has been put into creating a distributed controller, but most of the solutions provided by now have a monolithic approach that reduces the benefits of having a software defined network. Disaggregating the controller and handling it as microservices is the solution to problems faced when working with a monolithic approach. Microservices enable the cloud native approach which is essential to benefit from the architecture of the 5G Core defined by the 3GPP standards development organisation. Applying the concept of NFV allows to have a softwarised version of the entire network structure. The expectation is that the 5G Core will be deployed on an orchestrated cloud infrastructure and in this thesis work we aim to provide an application of this concept by using Kubernetes as an implementation of the MANO standard. This means Kubernetes acts as a Network Function Virtualisation Orchestrator (NFVO), Virtualised Network Function Manager (VNFM) and Virtualised Infrastructure Manager (VIM) rather than just a Network Function Virtualisation Infrastructure. While OSM has been adopted for this purpose in various scenarios, this work proposes Kubernetes opposed to OSM as the MANO standard implementation.

Key Words

SDN, NFV, Cloud Native, Kubernetes, 5GCore

1. Introduction

A technological growth in end user and industrial devices has brought to the production of a huge quantity of information, demand to optimised services and processing capabilities resulting in the necessity to bring innovation into the network infrastructure putting the basis to a new networking generation – 5G and paradigms such as Software Defined Networking (SDN) and Network Function Virtualisation (NFV). 5G is being seen as an opportunity for transformative change with the transition to cloud infrastructure and a service-based architecture.

Software defined networking allows to decouple the control plane from the data plane enabling the decentralisation of the controller handling it as collection of microservices hosted according to a cloud-native approach. There has been an ongoing effort to define some standard technologies to implement this concept but all of them present some lacks leaving space to search for alternative approaches. This thesis work proposes to host the core of 5G in a cloud native manner appointing Kubernetes as the orchestrator of the modules.

The following subsections go into more detail about the motivations behind the project work proposed in this thesis and the structure of the work that has been done.

1.1 Motivations

The fifth generation (5G) aims to put the basis for a revolution in the telecommunications industry thanks to its core characteristics such as extremely low latency, ubiquitous connectivity and the big improvement in terms of data-speed compared to its ancestor 4G. This allows to respond to an increasing demand of resources in a very broad scenario of applications. The enlarging number of mobile devices, wearable devices, demand of a reliable and fast connectivity everywhere, from massive Internet-of-Things to broadcast like services has created a gap that urges to be filled with a new generation of telecommunications. A key revolutionary aspect of 5G is to provide support for very heterogenous services in the same infrastructure even if the requirements of these diverse applications are very different from one another. There are three main scenarios for which the new generation has to provide services for: 1 – Ultra Reliable and Low Latency services (URLLC) 2 – Enhanced Mobile Broadband (eMBB) that focuses on services in need of high data rates 3 – Massive Internet-Of-Things (mIoT) consisting of a huge number of devices requiring connectivity i.e. smart cities.

Because of these heterogenous requirements the network trafficking is evolving into something much more complex compared to the past decades. Support for all these different scenarios is achieved by introducing the concept of Network Slicing. Network Slicing is a network architecture enabling the multiplexing of different independent and virtualised logical networks that share the same physical network infrastructure. Each slice is isolated from the others and is tailored to respond to specific requirements. To achieve such an infrastructure there are two main concepts that must be leveraged: Software Defined Networking (SDN) and Network Function Virtualisation (NFV) that enable the implementation of flexible and scalable network slices on top a common shared physical infrastructure.

Software Defined Networking is an architecture design that allows to make a network more flexible and easier to manage by centralising and abstracting the control plane from the data forwarding plane. An SDN architecture delivers a centralised and programmable network made of different components such as a controller - the core of the architecture responsible for centralised management and control, automation and policy enforcement – and a southbound API allowing a communication with the data plane made of switches, access points, routers etc. There is also a northbound API that allows the connection towards an application plane. This plane contains network applications that can introduce new network features, it can also receive an abstracted and global view of the network from the controllers and use that information to provide appropriate guidance to the control layer. SDN aims to change the concept of network seen so far by centralising the most difficult tasks to the control plane and making the data plane and its devices as simple as possible. Network Functions Virtualisation is a networking paradigm that proposes to shift from a traditional network infrastructure to a highly virtualised one where both the computing and networking services are deployed as virtual network functions. In other words, it is a softwarised implementation of network functions.

There has been a lot of effort in the telecommunication world towards the network virtualisation and after more than 100 specifications, ETSI has converged to a standard called MANO defining the guidelines for network virtualisation, management and orchestration of virtualised network functions (VNFs). One of the main implementations of MANO is also ETSI-hosted and is called Open Source MANO (OSM) – providing a programmable and automatic creation, management and orchestration of VNFs with support to various VIMs and SDNs. The 5G Core architecture has been designed to be cloud-native in the sense that it should make use of Network Functions Virtualisation and Software Defined Networking techniques and use service based interactions between control plane functions. A Service Based Architecture (SBA) aligns well with a microservices view of network function composition.

Most of the solutions provided so far on the deployment of 5G Core use OSM and different VIMs for the purpose of management, orchestration and hosting. The main idea of this thesis work is to find an alternative to the management and orchestration of the 5G Core components handling them as VNFs. The key concept is to use Kubernetes for these purposes and test the deployment under various scenarios. Kubernetes is an open source system for automating deployment, scaling, and management of containerized applications. The next section contains more details about the work, its structure and how this document will lay out the different steps.

1.2 Work Structure

To understand whether it is possible to use Kubernetes for 5G Core deployment and orchestration and how this can be achieved, it is necessary to understand very well the technology and its dependencies to have a fully working deployment. The core components of 5G have a very well-defined structure and require specific features from the hosting infrastructure so to function properly and maintain the interconnections between the modules. Hence, the first focus is to understand how to provide the requested properties with a very generic but widely spread technology like Kubernetes. It is necessary to gain knowledge about the technologies needed to have a successful deployment, how they could interact between them and perform the role of orchestrator.

These concepts are discussed in chapter 2 providing more information about the technologies at the basis of network softwarisation. It is also important to understand what we are hosting, so the structure of the 5G Core, how its components work and interact between them and which roles they cover. Chapter 3 contains this information, and it also lays out the differences of this generation of telecommunication compared to the previous one. Moreover, it contains references and description of the main implementations of 5G Core and a comparison between them so to better understand which is more suitable for this project. Chapter 4 lays out the infrastructure of the project and the interconnections between the technologies used for this project. The document then concludes with the achieved results and discusses future related work.

2. Network Softwarisation

This chapter starts by introducing the technologies that have enabled a softwarisation of the network such as SDN and VNF. It then moves towards the MANO standard and a big chunk of the chapter is dedicated to Kubernetes as it is the main technology used in this thesis work. Details about Kubernetes and 5G Core deployment on top of it are given in chapter 4, for now we will focus on a general overview of the technologies.

2.1 Software Defined Networking (SDN)

Software Defined Networking is an approach to network management that improves network performances and monitoring by enabling dynamic and programmatically efficient configuration of the network resulting in a more cloud computing like approach rather than a traditional network management one. SDN aims to resolve the issues related to the traditional static architecture of the networks putting the bases for a more flexible ones that responds to the requirements of the current complex telecommunications scenarios. In the traditional network architecture devices make traffic decisions and they forward data from one interface to another, so they perform the roles of both the control plane and data plane. Software Defined Networking centralises the network intelligence in a centralised network component by separating the forwarding process of network packets (data plane) from the routing process (control plane). The latter consist of one or more components called controllers considered to be the core of the SDN network where resides the whole intelligence of the infrastructure. The control plane is implemented as software allowing a programmatical and centralised organisation of the traffic. Another advantage that comes from this approach is to have data forwarding devices not requiring any intervention from the administrator to deliver the network services.

2.1.1 SDN Architecture

Figure 1 shows the architecture of a software defined network.

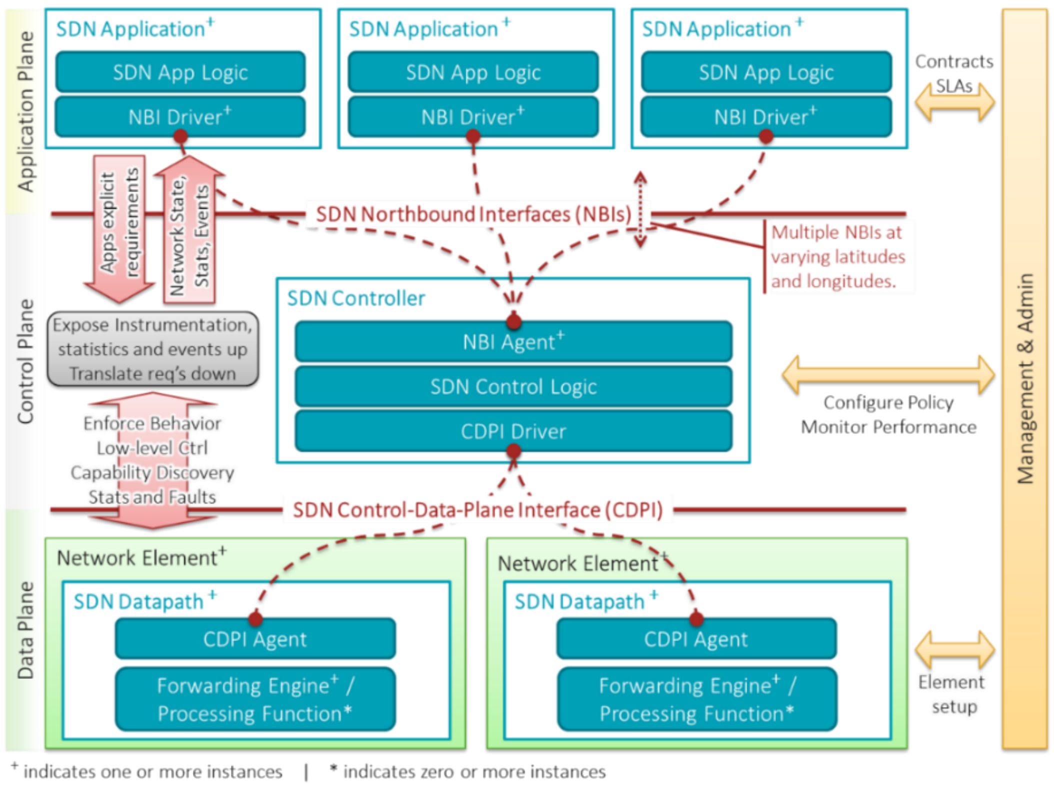


Figure 1 - SDN Structure

We have three main layers:

- ***Application plane*** – SDN Applications are programs that explicitly, directly, and programmatically communicate their network requirements and desired network behaviour to the SDN Controller via a northbound interface (NBI). In addition, they may consume an abstracted view of the network for their internal decision-making purposes. SDN Applications may themselves expose another layer of abstracted network control, thus offering one or more higher-level NBIs through respective NBI agents.
- ***Control plane*** - The SDN Controller is a logically centralized entity in charge of (i) translating the requirements from the SDN Application layer down to the SDN data paths and (ii) providing the SDN Applications with an abstract view of the network (which may include statistics and events). An SDN Controller consists of one or more NBI Agents, the SDN Control Logic, and the Control to Data-Plane Interface (CDPI) driver. Definition as a logically centralized entity neither prescribes nor precludes implementation details such as the federation of multiple controllers, the hierarchical connection of controllers, communication interfaces between controllers, nor virtualization or slicing of network resources.
- ***Data plane*** - responsible for processing data-carrying packets using a set of rules specified by the control plane. The data plane may be implemented in

physical hardware switches or in software implementations. The memory capacity of hardware switches may limit the number of rules that can be stored whereas software implementations may have higher capacity [1].

2.2 Network Functions Virtualisation and MANO

Network Function Virtualisation aims to decouple the network functions from the hardware by implementing them as a software package running on a virtualised infrastructure. This collapses multiple functions into a single physical server instead of installing expensive proprietary hardware, reducing in this way the costs and the dependency on dedicated hardware appliances enhancing scalability and customisation across the entire network. The following figure describes the main idea of NFV:

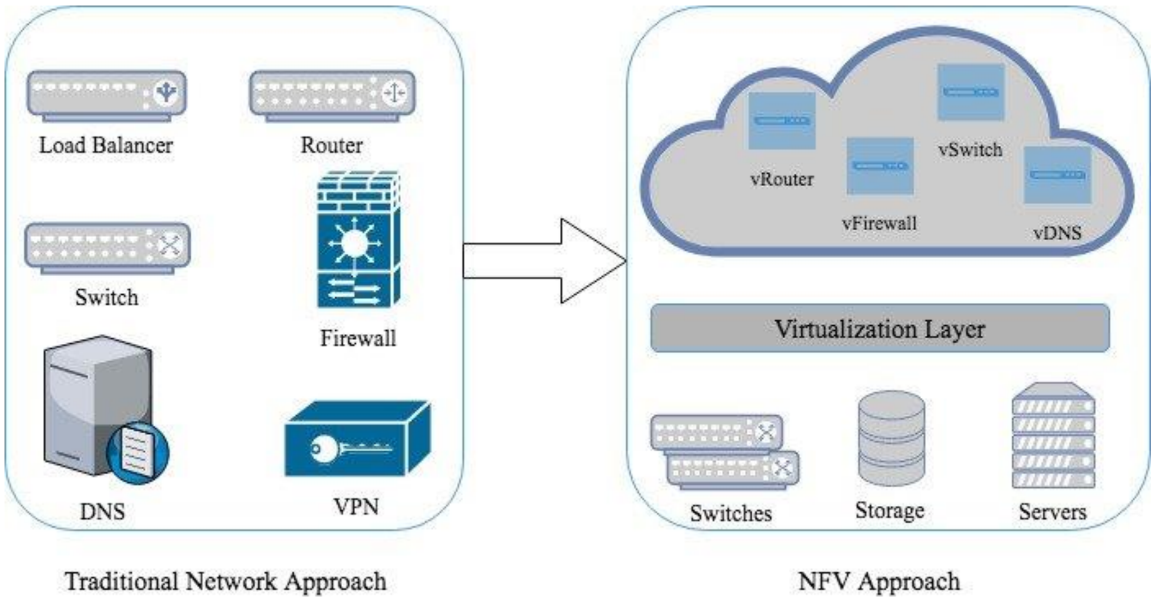


Figure 2 - Comparison Between Traditional Network Approach and NFV Approach

NFV application fields mainly include switching elements, mobile network nodes, traffic analysis, service assurance, content distribution networks (CDNs), security functions and session border controller (SBC), among others. NFV uses the virtualisation concept of three hardware resources: computing, storage and network hardware [2].

The below figure shows the NFV architecture:

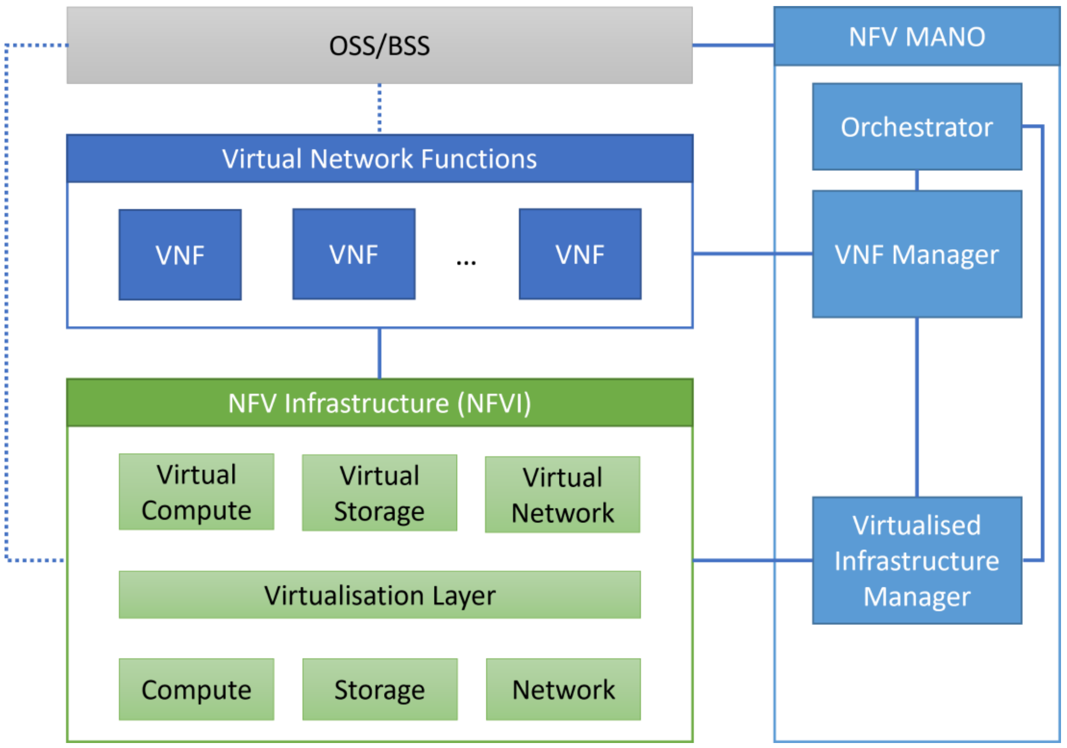


Figure 3 - NFV Architecture

As shown in the image, there are three main modules:

- NFV Infrastructure (NFVI) – it includes the hardware resources, the virtualisation layer and the virtual resources made of software instances of hardware resources such as computing, storage and network. The virtualisation layer abstracts the hardware resources and allows the deployment of virtual resources on top of it.
- Virtual Network Functions (VNFs) – software implementations of network functions running on the underlaying NFVI.
- NFV MANO – this module manages and orchestrates the VNFs and NFVI and has three elements: an orchestrator, a VNF Manager and a Virtualised Infrastructure Manager. The orchestrator is in direct contact with the application layer and manages the package on-boarding, network services lifecycle management, monitoring and fault management. The VNF manager instead handles the lifecycle of each active VNF in the network. The Virtualised Infrastructure Manager directly controls all physical resources, manages images, creation, modification and deletion of resource instances, monitoring and fault management.

2.2.1. Network Services in NFV

A network service can be viewed architecturally as a Forwarding Graph of Virtual Network Functions (VNFFG) interconnected by supporting network infrastructure. In other words, NS are a chain of VNFs and PNFs where PNFs are Physical Network Functions similar to VNFs but more connected to the physical layer [3].

Figure 4 shows the structure of an end-to-end network service:

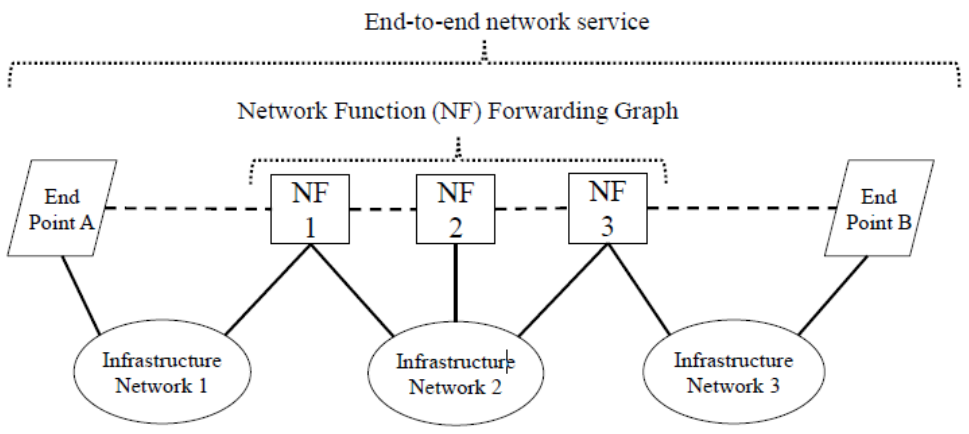


Figure 4 - End-to-end Network Service

The service illustrated in the above image has a nested Network Function Forwarding Graph, the endpoints are connected to the NFs via wired or wireless network infrastructure resulting in a logical interface (dotted lines) between the end point and a NF.

The deployment and operational behaviour requirements of each Network Service is captured in a deployment template and stored during the Network Service on-boarding process in a catalogue, for future selection for instantiation. The deployment template fully describes the attributes and requirements necessary to realize such a Network Service. Network Service Orchestration coordinates the lifecycle of VNFs that jointly realize a Network Service. This includes (not limited to) managing the associations between different VNFs, and when applicable between VNFs and PNFs, the topology of the Network Service, and the VNFFGs associated with the Network Service [4]. Network Services, when started, they require the deployment template but also the name of the service and a description. This results in a first phase of initialisation and configuration of the VFNs and the required Virtual Links. In this phase resources are allocated to enable a fully operating NS and the types of resources, and their interconnections depends on the underlaying VIM. When working with OpenStack, it will create virtual machines to host the VNFs, while working with Kubernetes will result in different Pods hosting VNFs. These concepts are very useful to perform a successful deployment of network functions on Kubernetes, more details in chapter 4.

The whole NFV infrastructure is the definition of a concept that is open to implementation, anyone willing to provide their own solution as implementation of these components can do so and provide a way to deploy, manage and orchestrate VNFs. ETSI has indeed provided a standard for management and orchestration called MANO along with an ETSI-hosted implementation of this standard called Open Source Mano (OSM). In the framework of this thesis work, the goal is to understand whether Kubernetes can be used to perform the role of the Virtualised Infrastructure Manager and Orchestrator. The next chapter provides a detailed introduction about Kubernetes and its main characteristics while its usage in the implementation will be discussed in chapter 4.

2.3 Kubernetes

This section lays out a detailed introduction to Kubernetes, the current industry de facto standard container orchestrator. The below subsections contain information about what Kubernetes is, how it works, its architecture and components. Kubernetes has a wide range of components but here we will only discuss the ones necessary for this thesis work.

2.3.1 Introduction to Kubernetes

Kubernetes, also referred to as K8s, is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.With K8s it is possible to orchestrate and automate the deployment, scaling, load balancing of containers along with grouping them into logical units [6].

The use of a container orchestrator is very beneficial in complex scenarios in which handling manually the instantiation and subsequent lifecycle management of containers is not a trivial task. Indeed, the container communication is not an easy task, scaling them to provide the required services can be very complex to manage manually, the same applies when a node crashes and needs to be fixed. Despite the benefits coming from applications containerization, when the number of containers goes up considerably it becomes very difficult to manage the portability aspect of that services in case of any fault.

Therefore, container orchestration becomes essential for an effective management of the infrastructure and Kubernetes provides a framework to run distributed systems resiliently. It takes care of scaling and failover for the hosted applications, along with providing deployment patterns, and much more.

Kubernetes main features:

- *Service Discovery and Load Balancing* – containers are exposes using the DNS name or their IP address. If a certain container has a high load of traffic towards it, K8s can load balance and distribute the network traffic so that the deployment is stable.
- *Storage Orchestration* – it is possible to mount any storage system, be it local storage, remote storage or public cloud providers.
- *Automated Rollouts and Rollbacks* – Kubernetes allows to provide a description of the desired state of deployed containers and then it will manage the cluster so to change the actual state to the desired state at a controlled rate.
- *Automated bin packing* – it is possible to provide K8s with a cluster of nodes that can be used to run containerised tasks. When it receives the instructions on how much CPU and RAM it can use, then K8s will fit the container on the available resources in a such a manner that allows the best usage of resources.
- *Self-healing* - Kubernetes restarts containers that fail, replaces them, kills containers that don't respond to the user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- *Secret and configuration management* - Kubernetes allows to store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. It is possible to deploy and update secrets and application configuration without rebuilding the container images, and without exposing secrets in the stack configuration.

Kubernetes is not a traditional PaaS system, but it provides some features common to PaaS offering as it works at the container level rather than at a hardware level. These features include the above-mentioned deployment, load balancing, scaling but also logging, monitoring and alerting solutions. However, Kubernetes is not monolithic, it lets the user decide what he requires the most preserving flexibility where it is important. There is not a limit to the type of application that K8s can support, be it a stateless, stateful or data processing workload Kubernetes will support it. The main idea is that if an application can run in a container, then it should run perfectly also on Kubernetes.

2.3.2 Kubernetes Components

A successful deployment of Kubernetes creates a cluster that is made of a set of worker nodes that can run containerised applications. Each cluster consists of at least one worker node. These nodes host the pods that are the components of the application workload, we will discuss these later. All the nodes and pods in the cluster are managed by the control plane. Below images shows the structure of a Kubernetes cluster:

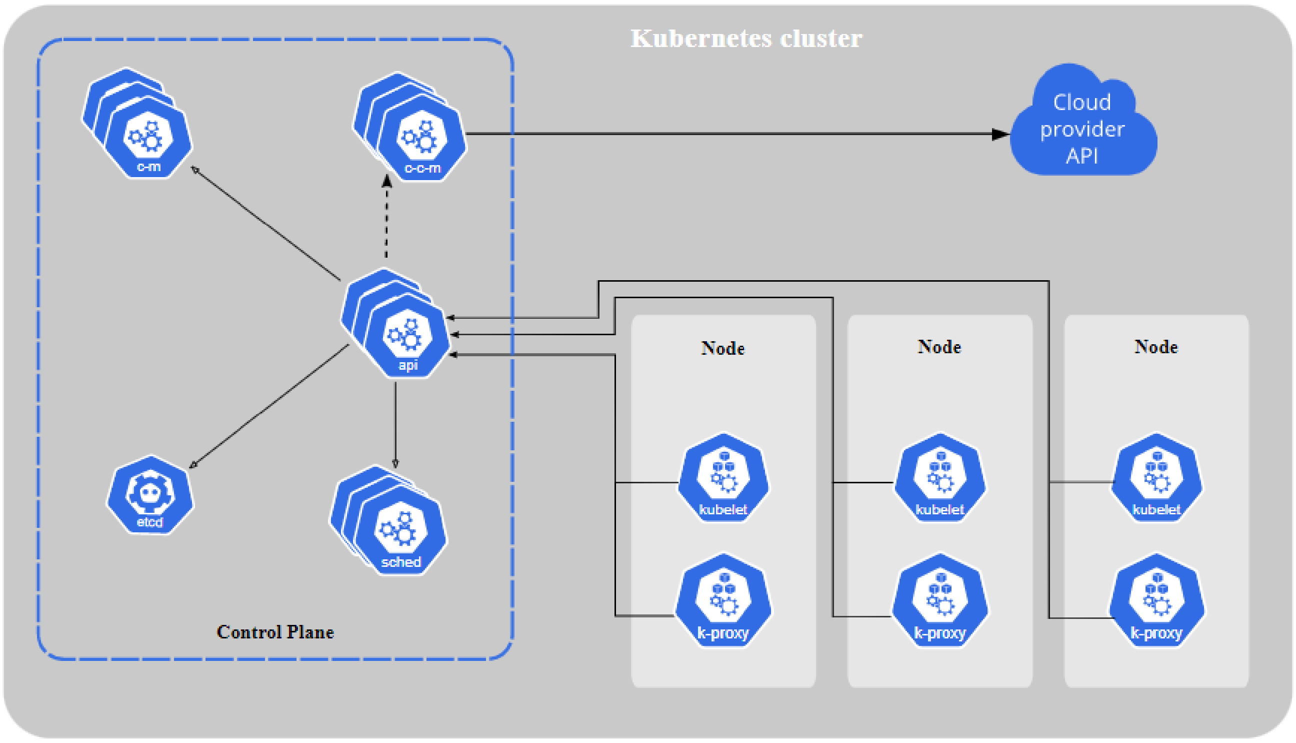


Figure 5 - Kubernetes Cluster

Control Plane Components

The components of the control plane make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied). These components can be run on any machine in the cluster, however, for simplicity they are put on the same machine.

- **Kube-apiserver -** exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. The main implementation of a Kubernetes API server is kube-apiserver. Kube-apiserver is designed to scale horizontally so it scales by deploying more instances. It is possible to run several instances of kube-apiserver and balance traffic between those instances.

- **Etcd –** it is a consistent and highly-available key value store that is used as a backing store for all the cluster data. If the cluster uses etcd then we also need a back up plan for this data as K8s does not handle data persistency.
- **Kube-scheduler –** this component observes for newly created Pods with no assigned node and selects a node for them to run on. For scheduling, it takes into consideration factors such as individual and collective resource requirements, constraints, data locality, inter-workload interference and deadline.
- **Kube-Controller-Manager –** component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. Some examples of main controller are node controllers and job controllers.
- **Cloud-Controller-Manager –** it embeds cloud-specific control logic. It allows to link the cluster to a cloud provider's API, defines a separation between the components that interact with the cluster and the ones that interact with that cloud provider. When running Kubernetes on premises, this component is not instantiated.

Node Components

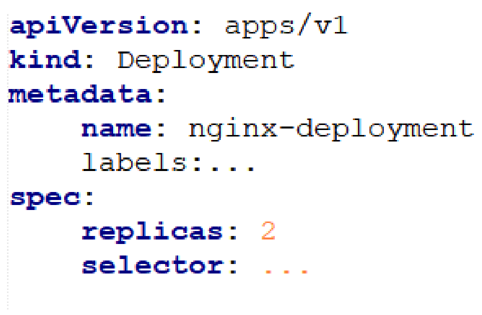
- **Kubelet -** An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy.
- **Kube-proxy -** it is a network proxy that has an instance running on each node of the cluster implementing part of the Kubernetes Service concept (services are discusses later in this chapter). This proxy maintains network rules on the node and these rules allow the communication to the Pods from network sessions residing either inside or outside the cluster.
- **Container runtime** – components responsible for running containers that will be hosted inside Pods.

Kubernetes provides a command line tool for communicating with a Kubernetes cluster's control plane using the Kubernetes API. This tool is called ***kubectl.***

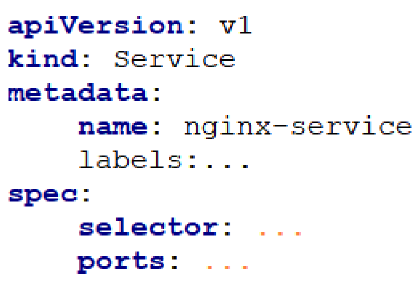
2.3.3 Configurations

Configuration files are the main tool for creating and configuring components in Kubernetes Cluster. In fact, K8s objects can be created updated, and deleted using the configuration files and *kubectl apply* command. Configuration files are in YAML format and have a well-defined structure.

The images below show two simple configuration files for two different cases - a deployment and a service (these concepts will be discussed later):



```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    labels:...
spec:
    replicas: 2
    selector: ...
```



```
apiVersion: v1
kind: Service
metadata:
    name: nginx-service
    labels:...
spec:
    selector: ...
    ports: ...
```

Each configuration file has three parts:

- Metadata – here it is possible to define the name and eventually some labels. This section defines what we are intending to create. In the image at the left there is a Deployment while on the right there is a Service.
- Specification – contains all the configurations that we aim to provide for that specific component that we are creating. The attributes are specific to the kind of component that we are creating.
- Status – the desired one is defined by the specification section while the actual status of the cluster is automatically generated by Kubernetes. The desired status of the applications is defined by the admin in the specifications, and then K8s compares this with the actual status of the cluster. If these two are not equal, then Kubernetes tries to fix it and this is the basis of the self-healing features discussed in the Kubernetes introduction. In the above example, when creating the Deployment, the file defines that the desired state requires 2 replicas of that Deployment. Then K8s, as it updates the state continuously, will know if there are not 2 replicas and, in that case, it will intervene and enforce the desired status. Information about the status is stored in the etcd component, introduced in the introduction section.

2.3.4 Workloads and Services

A workload can be defined as an application running on Kubernetes. Whether the workload is a single component or several components interacting and working together, on Kubernetes they run inside a set of pods. A Pod represents a set of running containers on your cluster and provides an abstraction over these containers. This subsection goes into more details about the main components of Kubernetes such as Pods, Services, and ConfigMaps.

Pods

Pods are the smallest computing unit that can be deployed and managed in a Kubernetes cluster. It can be seen as a group of one or more containers that have storage and network resources in common and a specification for how to run the container. A Pod's contents are always co-located and co-scheduled and run in a shared context. As said before, a pod provides an abstraction over the containers it hosts inside. These containers run, of course, in a container runtime environment and the main one that is used in Kubernetes is Docker, even if K8s supports also others such as Containerd, CRI-O and Mirantis. In terms of Docker concepts, a Pod can be seen as a group of Docker containers with shared namespaces and filesystem volumes.

Usually there is a “one pod – one container” policy but this can vary depending on the specific contexts. When it runs more containers then usually there is an idea of maintaining together the tightly coupled ones. So, in general there are two main ways to use Pods in a cluster:

- *Pods that run a single container* - the "one-container-per-Pod" model is the most common Kubernetes use case; in this case, it is possible to think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.
- *Pods that run multiple containers that need to work together* - a Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service — for example, one container serving data stored in a shared volume to the public, while a separate container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.

The below image shows the structure of a node and an example of a node that contains a set of Pods and each of them can either be a single container Pod or multiple containers one:

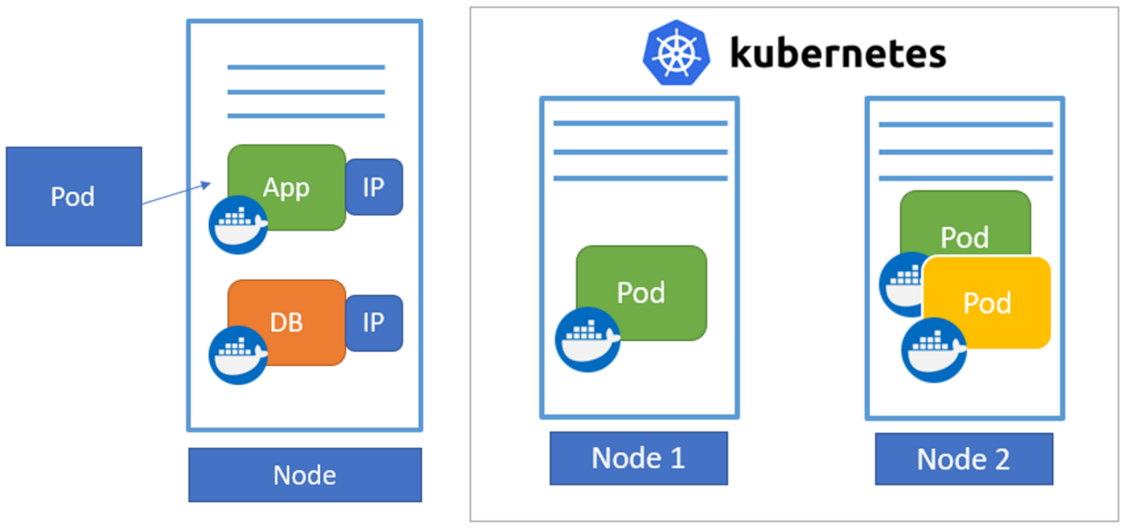


Figure 6 - Example of a Node

When a Pods is created, it is scheduled to run on a node and it remains there until it finishes execution, it is deleted, evicted or the node fails. Pods are designed as ephemeral and disposable entities, and this is the reason why it is very rare to interact with them directly when deploying an application over a K8s cluster. Workload resources can be used to create and manage multiple Pods and the resource controller handles replication, rollout and automatic healing in case a Pod fails. For example, if a Node fails, a controller notices that Pods on that Node have stopped working and creates replacement Pods. The scheduler places the replacement Pod onto a healthy Node.

Kubernetes provides several built-in workload resources:

- *Deployment* – allows to manage a stateless application workload on a cluster. The idea being that each Pod does not maintain any state so it is entirely interchangeable with another Pod of the deployment and can be replaced if needed.
- *StatefulSet* – runs one or more related Pods that track state. If the workload records data persistently then a StatefulSet can be used and it matches each Pod with a PersistentVolume (this will be discussed later in the chapter). The application running in the Pods of a StatefulSet can replicate data to other Pods running in it to improve overall resilience.

- *DaemonSet* – it defines Pods that provide node-local facilities, and these are fundamental for the cluster to run properly. If the control plane detects a node that is added to the cluster and matches the specification of a DaemonSet, it schedules a Pod for that DaemonSet onto the new node.
- *Job and CronJob* – define tasks that run until completion and then stop. Jobs represent one-off tasks while CronJobs are reoccurring ones depending on a schedule.

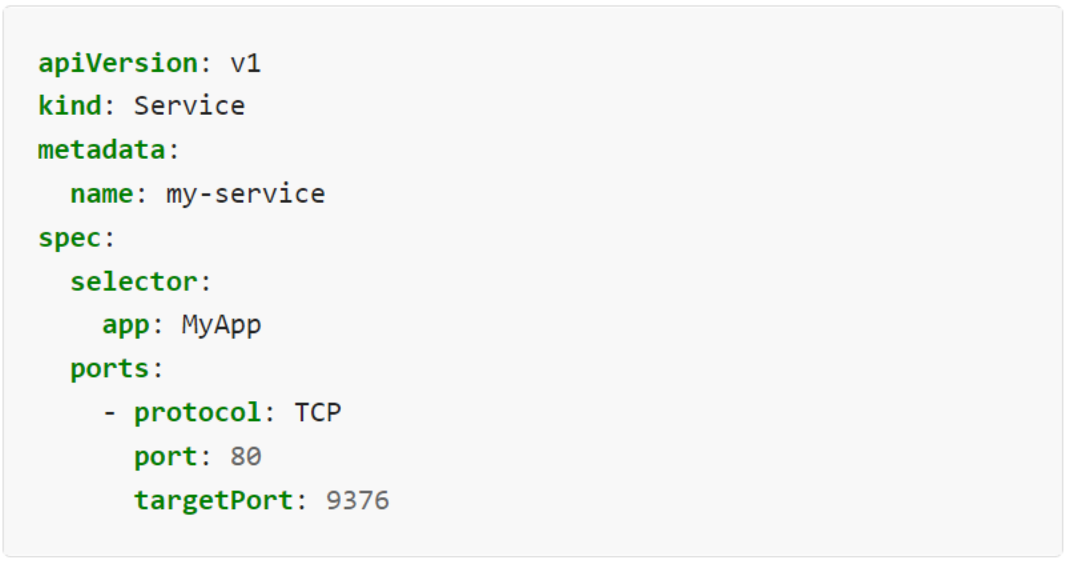
Every Pod in the cluster is assigned an IP address that is unique inside the cluster which means that there is no need to create explicit links between Pods or manage container ports to host ports mapping. Here the concept it to think about Pods treated like Virtual Machines or physical hosts from the perspective of port application, naming, service discovery, load balancing, application configuration and migration. Kubernetes IP addresses exist at the Pod scope - containers within a Pod share their network namespaces - including their IP address and MAC address. This means that containers within a Pod can all reach each other's ports on localhost. Whenever a Pod is deleted, for example as a result of a failure, it is replaced by another Pod but this one will have an IP address different than the previous one and this affects the communication and service delivery of the application running inside that Pod. For this reason, and to enable service delivery both inside and outside of the cluster, Kubernetes introduces the concept of Services.

Services

Services are an abstract way to expose an application running on a cluster as a network service. As mentioned before, Pods are created and destroyed because of many different reasons, one of the main ones is to match the desired state of a cluster. They are not considered permanent resources, Pods running in the cluster on a specific moment may be very different from the ones running in a successive moment. At the creation time of a Pod, it is given a unique IP address which leads to a problem: if a set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside the cluster, how do the frontends find out and keep track of which IP address to connect to, so that they can use the backend part of the workload? The concept of services overcome this issue.

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service). Suppose to have a stateless image-processing back-end which runs three replicas which are interchangeable – front-end can work with any of these three to work. If the Pods that compose the back-end change, the front-end clients should not be aware of that change nor they should need to keep track of it. This requires a decoupling that is provided by the Services.

A Service is a REST object, similar to a Pod, so it is possible to POST a Service definition to the API Server within the cluster to create a new instance. Services can be defined using the configuration files in YAML format, the below image shows a possible configuration file to create a Service:



```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Figure 7 - Web Service Configuration File

Suppose to have a set of Pods containing an application MyApp and listening on a TCP port 9376, this specification file created a service which target TCP port 9367 on any Pod with the label app=MyApp. The Kubernetes assigns this Service an IP address which is used by the Service proxies.

When creating a Service, this can be either an internal service or an external one. The internal services can be used for example to interact with a database, which should be exposed internally within the cluster but for obvious reasons not accessible from the outside. This changes when we need to expose some functionalities towards the external world, and this results in the need to of having an external service that is reachable from the outside. The external communication is managed by another component called ***ingress.*** Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource. The following image shows an ingress sending all its traffic to one Service:

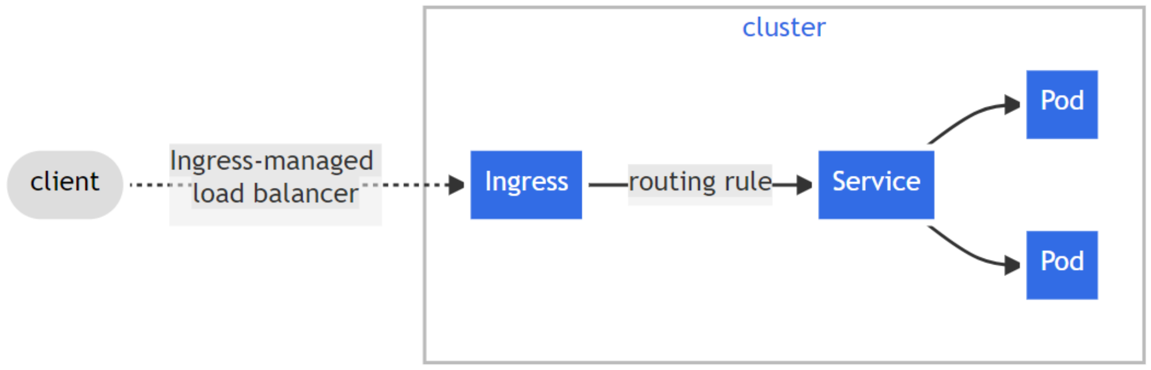


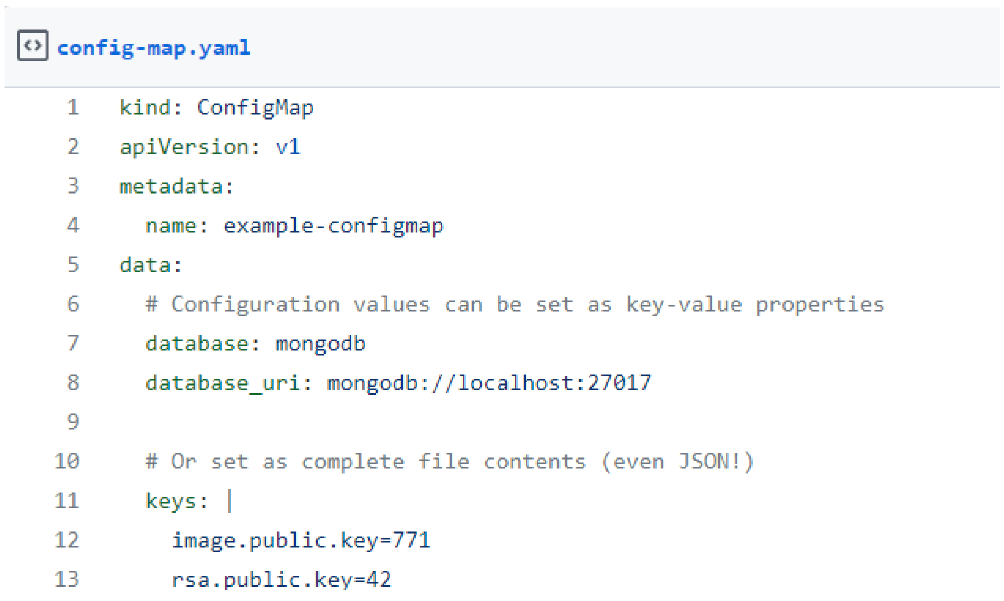
Figure 8 - Kubernetes Ingress

ConfigMap

When developing an application that will be deployed on a Kubernetes cluster, there are several configurations that can be set, for example the address to the mongo-db. Usually these configurations would be set in an external file, or an external environmental variable but even if these are external, they would still be part of the build. This becomes a problem when there are small changes that need to be made and they result in a work overload as every time they would require to re-build the application, push it to the Docker repository and then pull it again in the Pod. This becomes a little tedious so for this purpose Kubernetes has a component called ConfigMap.

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume. It allows to decouple environment-specific configuration from container images, so that the applications are easily portable. When writing a Pod spec it is possible to refer to a ConfigMap and configure the container(s) in that Pod based on the data contained in the ConfigMap.

When a ConfigMap currently consumed in a volume is updated, projected keys are eventually updated as well. The kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. The following snippet of code shows an example of ConfigMap with the value database set to mongodb, and database_uri, and keys set to the values in the YAML example code:



config-map.yaml

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: example-configmap
data:
  # Configuration values can be set as key-value properties
  database: mongodb
  database_uri: mongodb://localhost:27017

  # Or set as complete file contents (even JSON!)
  keys: |
    image.public.key=771
    rsa.public.key=42
```

Figure 9 - structure of a ConfigMap

2.3.5 Storage

An application running on a Kubernetes cluster generates data that would be lost in case there is a restart of the Pod or the container. Files in a container are ephemeral which becomes a problem for non-trivial applications. When restarted, the kubelet restarts the container with a clean state. More problems occur when sharing files between containers running together in a Pod. In order to avoid these issues, it is necessary to make the data persistent and to do so Kubernetes introduces the concept of ***Volumes***. This allows to attach a physical storage resource to the Pods, and it can be either local to the cluster or external, for example on a remote storage or cloud. The below image shows this set up:

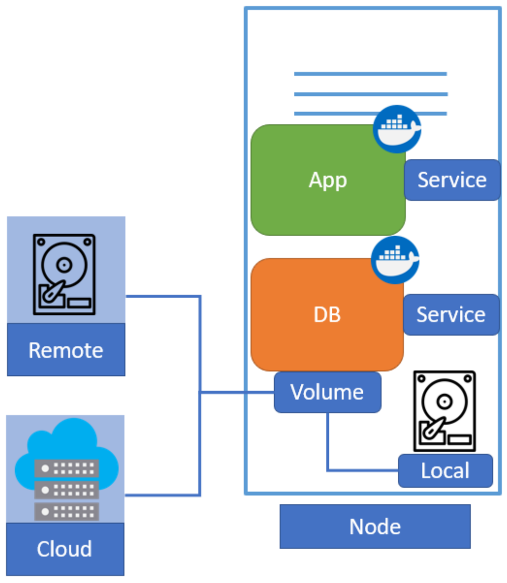
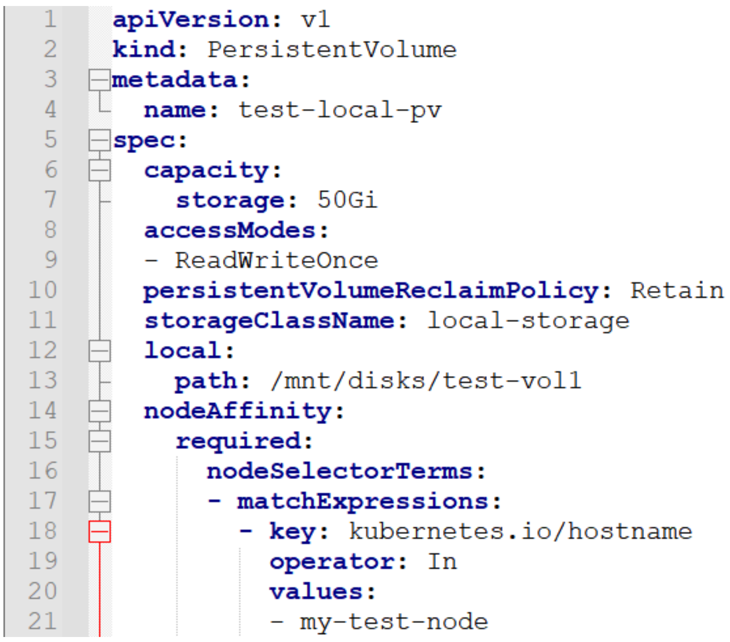


Figure 10 - Storage in a Kubernetes Node

Kubernetes itself does not manage any data persistency, it is up to the developer or admin to manage, back-up and replicate the data in a persistent way. Kubernetes Volumes allow to have data persistency and many types of volumes are supported. When a Pod ceases to exit, Kubernetes destroys the ephemeral volumes, however the persistent volumes are kept intact. Volumes can be seen as directories accessible to the containers in a Pod. Managing storage is a distinct problem from managing compute instances. For this purpose, Kubernetes has introduced the *PersistentVolume* subsystem that provides an API for user and admins allowing to abstract details of how the storage is provided from how it is consumed. There are two API resources for this: *PersistentVolume* and *PersistentVolumeClaim.*

A PersistentVolume (PV) is a piece of storage that has been provisioned either from the admin or dynamically. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage or a cloud-provider-specific storage system. A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany).

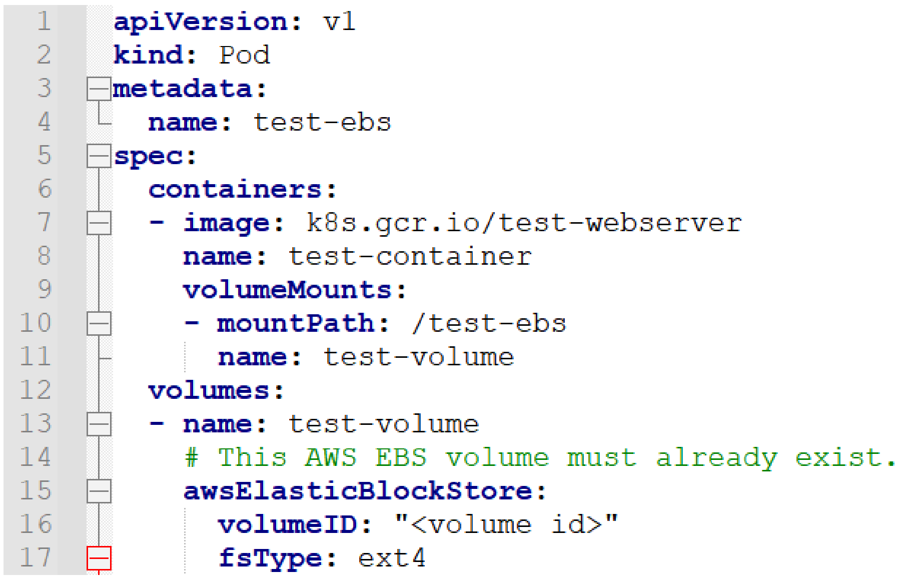
As mentioned earlier, K8s supports several types of volumes, either local or for cloud, i.e. AWS EBS. The images below show examples of local (first image) PersistentVolume creation and AWS EBS configuration example (bottom image):



```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-local-pv
spec:
  capacity:
    storage: 50Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  local:
    path: /mnt/disks/test-vol1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - my-test-node
```

Figure 11 - Local Persistent Volume

This YAML file creates a local volume of 50Gi with a ReadWriteOnce access mode. There is also a *persistentVolumeReclaimPolicy* that defines the policy for PersistenVolumeClaim. A PersistentVolumeClaim is used to mount a PersistentVolume into a Pod. PersistentVolumeClaims are a way for users to "claim" durable storage without knowing the details of the environment. (Details discussed later in this subchapter).

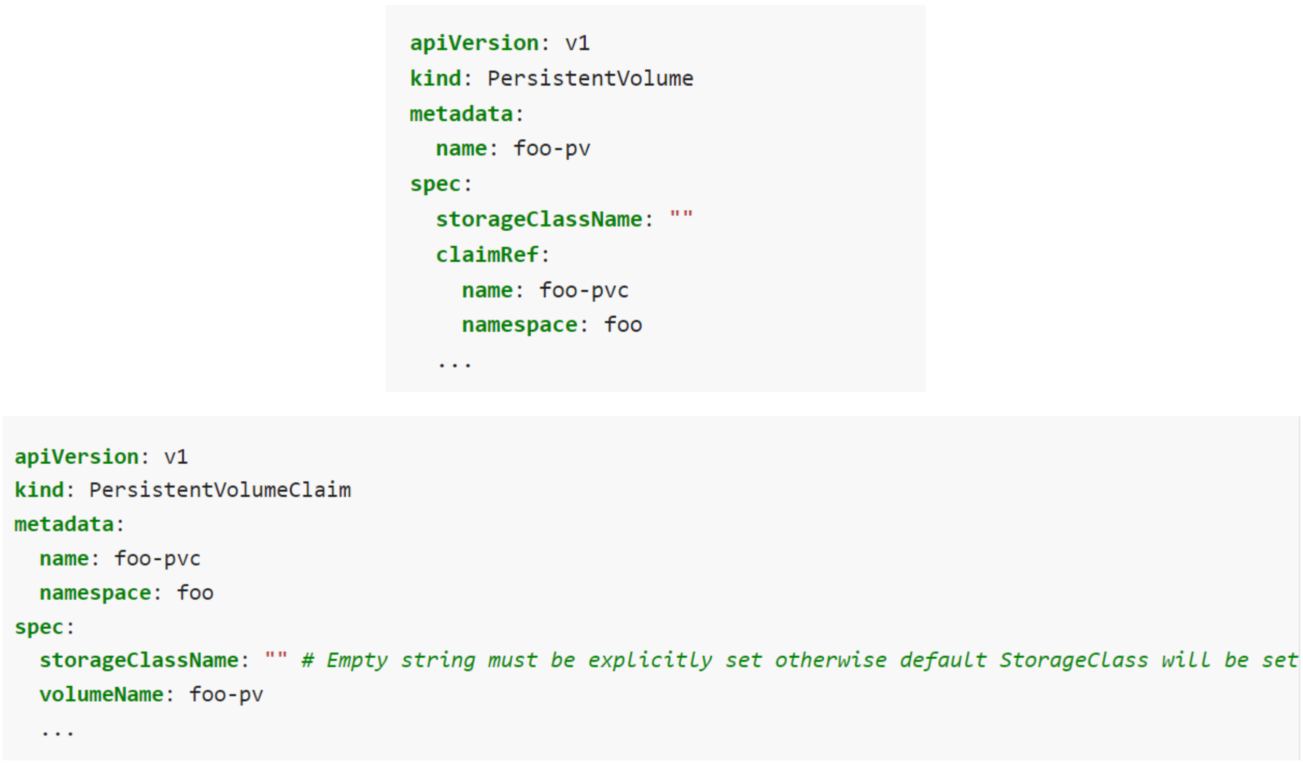


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: "<volume id>"
      fsType: ext4
```

Figure 12 - AWS EBS Persistent Volume

Here instead we have a volume on the cloud (AWS) and when we create this in the cluster, the AWS EBS must already exist beforehand to avoid failures.

A binding between PVs and PVCs is needed for the storage to work properly. By specifying a PersistentVolume in a PersistentVolumeClaim, it is possible to declare a binding between that specific PV and PVC. If the PersistentVolume exists and has not reserved PersistentVolumeClaims through its claimRef field, then the PV and PVC will be bound:



```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: foo-pv
spec:
  storageClassName: ""
  claimRef:
    name: foo-pvc
    namespace: foo
  ...
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: foo-pvc
  namespace: foo
spec:
  storageClassName: "" # Empty string must be explicitly set otherwise default StorageClass will be set
  volumeName: foo-pv
  ...
```

Figure 13 - Definition of a PersistenVolume and PersistenVolumeClaim

A volume, in its lifecycle can be in one of the following states:

- Available – free resource, not yet bound
- Bound – bound to a claim
- Released – the claim has been deleted and the resource is not reclaimed by the cluster
- Failed – the volume has failed its automatic reclamation

Each PV has its own set of access modes, the ones supported in Kubernetes are:

- ReadWriteOnce – read-write by a single node (so Pods running on that node)
- ReadOnlyMany – read only by many nodes
- ReadWriteMany – read write by many nodes
- ReadWriteOncePod – read write by a single Pod

After this introduction to the technologies that have enabled the network softwarisation, the next chapters go into more details about 5G Core, its implementation and then towards the deployment of the Core on a Kubernetes cluster. More technologies used to achieve this goal will be described in the chapter 4.

3. 5G Core – Structure and Implementations

Fifth generation (5G) networks respond to the demands of a wide range of heterogeneous use cases, including low latency services, mobile broadband users and highly dense connectivity scenarios. These very diverse communication contexts require a networking with a huge degree of flexibility, something not found in the previous technologies with a black-box approach like 4G cellular networks. The potential of 5th generation (5G) communications is being unleashed into the fabric of cellular networks, enabling unprecedented technological advancements in the networking hardware and software ecosystems [6][7]. This chapter is a detailed description of the technology, the motivation that brought to the next generation, the core of 5G and its implementations.

3.1 Motivation

The previously mentioned demand of resources and services stems from an increase in the number of devices that the technological world has experienced in the last few years. The amount of these devices is predicted to rise in the near future in an exponential way. Referring to only the IoT devices, it has been estimated that the current 10 billion of these devices will surpass 25 billion by 2030. And by 2025 there will be more than 152.000 IoT devices connecting to the Internet *per minute* [9].

The quantity of resources needed, and the amount of data produced becomes even higher when considering all technological devices together and not limiting to just the IoT ones. The introduction of new applications to satisfy the demands of customers all over the globe requires a re-design of the network infrastructure if the requirements differ substantially from what is supported. The evolution from the previous generations reaching now the fifth generation has been driven by the evolution of the supported applications and subsequently the new demand in services.

5G needs to support the traditional human-type applications such as high-speed video streaming, VoIP etc. but it is not limited to that. In fact, it must enable a wide range of applications coming from autonomous and communication among sensors and actuators in scenarios like Industry 4.0 or Smart Cities. Intelligent mobility is another field of application along with remote control which is another service that allows to remotely control objects that are connected to the network. This concept can be applied in many scenarios such as smart cities and health care industry.

The following image shows the evolution of the networks starting from 1G to 5G and the supported applications over the generations:

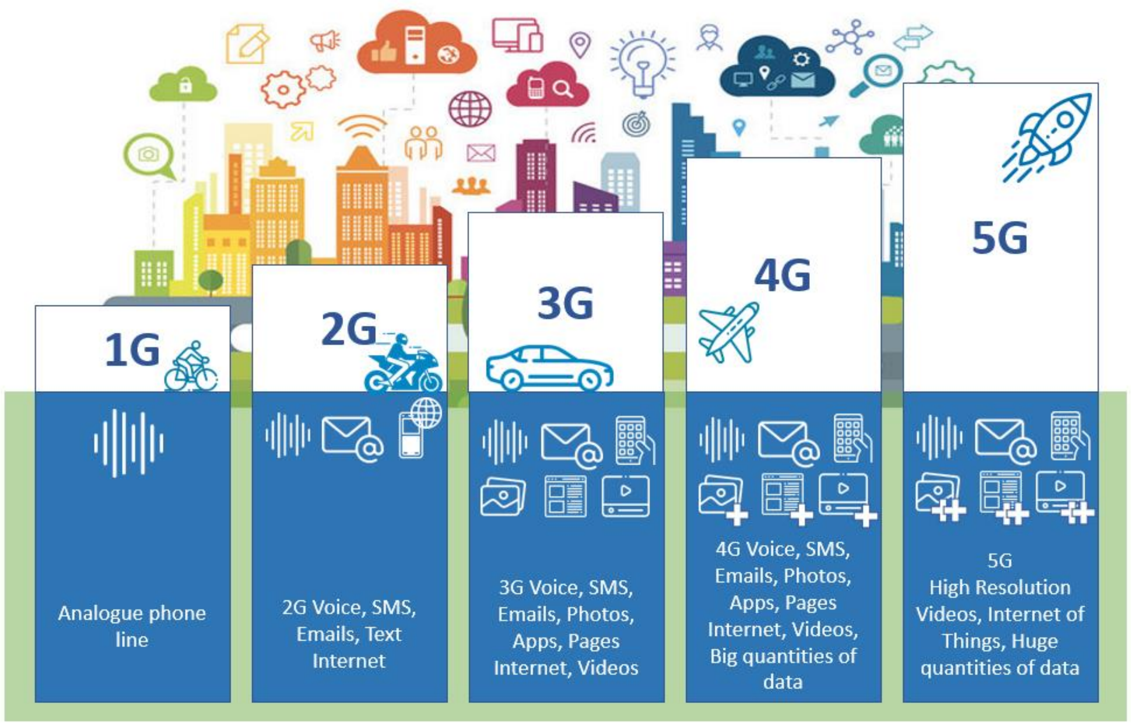

1G
2G
3G
4G
5G
Analogue phone line
2G Voice, SMS, Emails, Text Internet
3G Voice, SMS, Emails, Photos, Apps, Pages Internet, Videos
4G Voice, SMS, Emails, Photos, Apps, Pages Internet, Videos, Big quantities of data
5G High Resolution Videos, Internet of Things, Huge quantities of data

Figure 14 - Network Generations Evolution

- 1G – it supported mobile phone and voice only
- 2G – mobile phone and text messages [2.5 data connectivity over mobile networks through Wireless Application Protocol (WAP)]
- 3G – mobile phone and GPS
- 4G – mobile phone and minimal video streaming
- 5G – mobile phone and streaming, IoT, AI, VR, AR

The shift from 2G to 3G was mainly to introduce mobile Internet and some additional functionalities to the pre-existing network. The architecture of 2G, also known as GSM, was made of entities that all managed both the traffic (user plane) and traffic control (control plane). 2.5G introduced data connectivity for mobile networks and 3G mainly focused on overcoming the poor channel capacity of 2.5G. 4G was mainly motivated by the need of high-speed data connectivity for mobile web browsing as well as to support real-time Internet traffic i.e. video streaming.

4G presents some limitations from the architectural perspective. The static deployment of vendor equipment and a use of monolithic functionality at specific network locations inherited from the previous generation are some of the reasons behind the limitations presented by the network infrastructure. The negative aspects of 4G are:

- *Inflexibility* – due to the static deployment of vendor equipment as mentioned before but also because of the use of proprietary black boxes and monolithic functionalities. Updates require the replacement of existing equipment even if still fit for purposes it serves.
- *Complexity* – network entities are involved both in user plane and control plane.
- *Centralised User and Control plane* – this results in a waste of network resources even if UEs do not have data to send and there is also a reduction of the overall capacity of the network.

5G aims to overcome the limitations of the fourth generation along with providing support for the diverse requirements needed by the scenarios previously mentioned. Its capabilities are mainly:

- *Flexibility and programmability* – allowing fast reconfiguration, resiliency and a dynamic network topology.
- *Scalability* – enabling service aware QoS, low latency and availability supporting massive numbers of devices. Scalability also in terms of ease in integration and interoperability with other technologies and protocols.
- *Differentiated management of user and control plane* – planes are decoupled, it provides slice-based control plane and user plane functions enabling ultra-low latency, QoS management and reliability.

Thanks to the above-mentioned characteristics, 5G networks is able to support the diverse scenarios as mentioned before and shown in the below image:

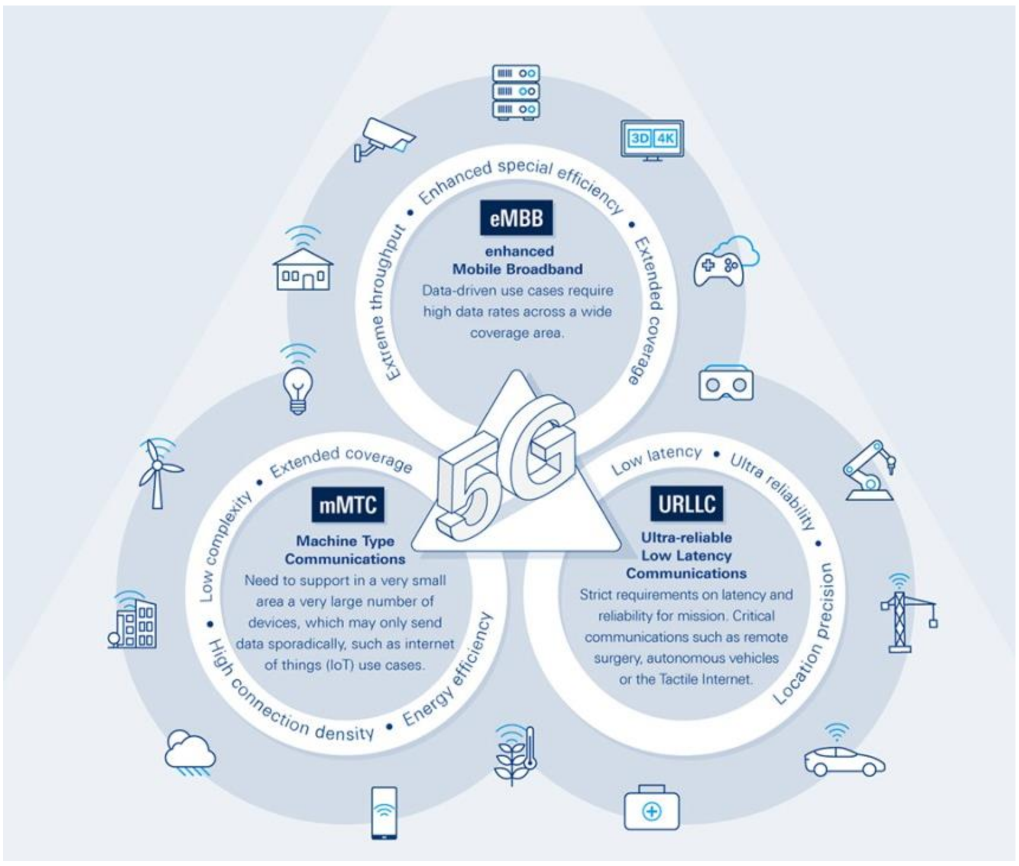

eMBB
enhanced Mobile Broadband
Data-driven use cases require high data rates across a wide coverage area.
Enhanced special efficiency
Extended coverage
Extreme throughput
mMTC
Machine Type Communications
Need to support in a very small area a very large number of devices, which may only send data sporadically, such as internet of things (IoT) use cases.
Extended coverage
Low complexity
High connection density
Energy efficiency
URLLC
Ultra-reliable Low Latency Communications
Strict requirements on latency and reliability for mission. Critical communications such as remote surgery, autonomous vehicles or the Tactile Internet.
Low latency
Ultra reliability
Location precision
5G
3D 4K

Figure 15 - 5G Main Characteristics

Enhanced Mobile Broadband (eMBB) aims to serve more densely populated urban centres with downlink speeds approaching 1 Gbps indoors and 300 Mbps outdoors. Massive Machine Type Communications (mMTC) enables machine-to-machine (M2M) and Internet of Things (IoT) applications that a new wave of mobile customers could expect from their network without burdening the other classes of service. Ultra-Reliable and Low Latency Communications (URLLC) would address critical communication needs where bandwidth is not as important as speed, i.e., end-to-end latency of 1 ms or less [12].

3.2 Evolution of the Network Infrastructure

5G has brought a very different approach to the network infrastructure as it changes the way components are deployed compared to the 4G. However, 5G will be initially in a non-standalone mode and it will work in parallel to the 4G LTE network. Over the next few years this will change as more and more the 5G network infrastructure will expand enabling a standalone mode. The below image shows these two modes:

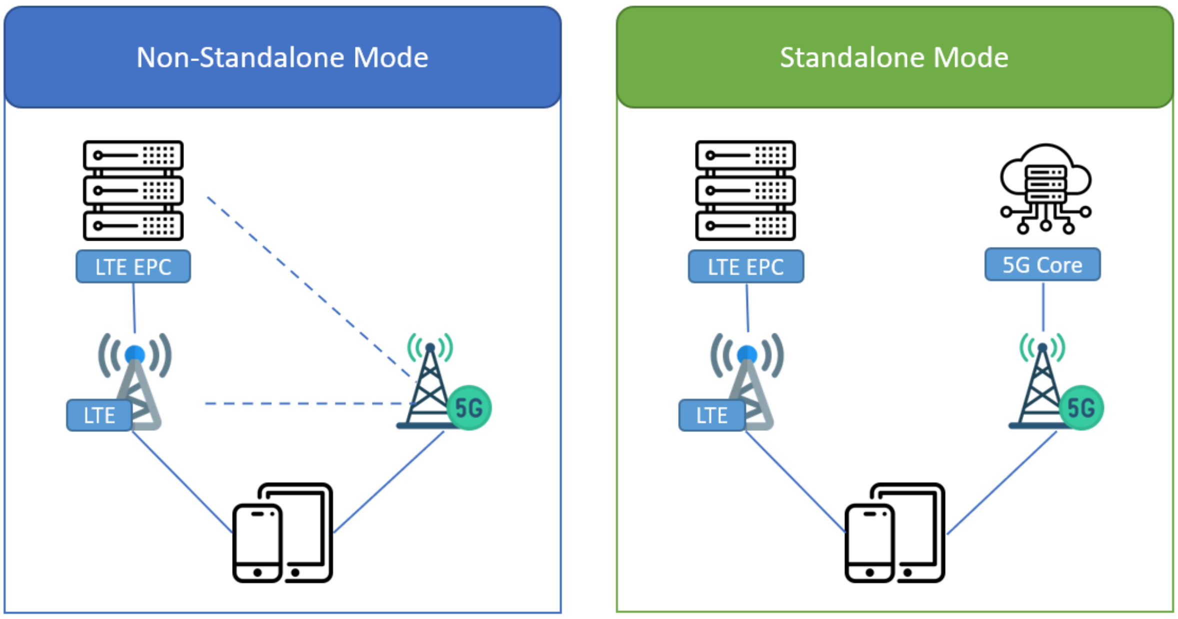


Figure 16 - Standalone vs Non-Standalone Mode

In a non-standalone mode, a mobile device connecting to the network will first connect to the 4G and then if 5G is available it will use it for additional bandwidth. Whereas in the standalone mode the whole infrastructure will be separated.

The below image allows to understand how different these two networks in terms of deployment components distribution and coverage are:

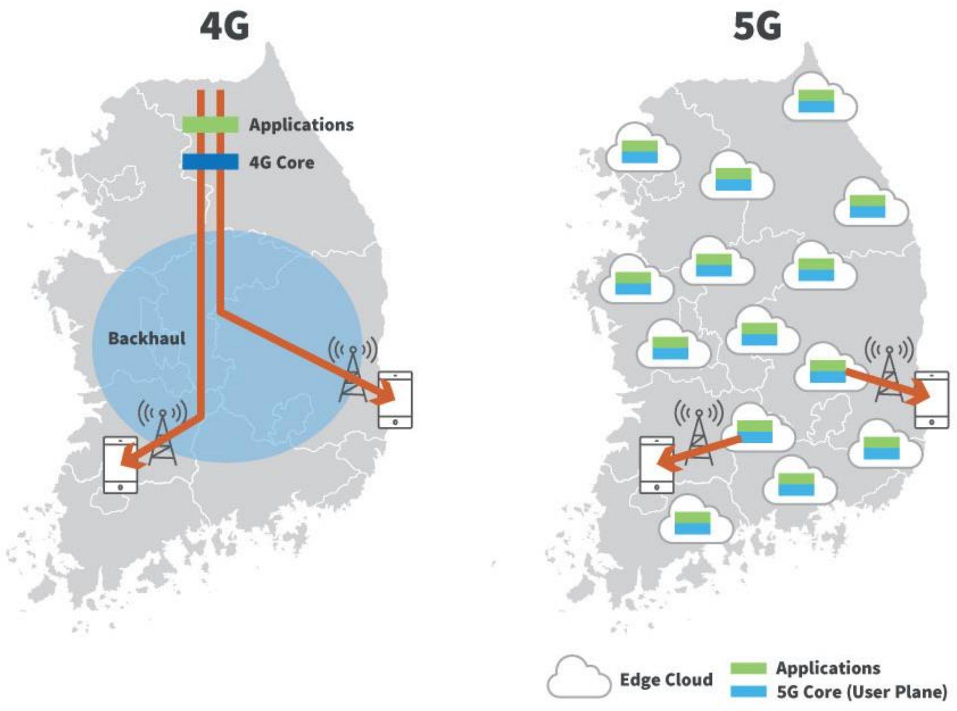


Figure 17 - 4G and 5G Infrastructure Comparison

In 4G LTE network architecture the RAN and eNodeB are typically close together and at the base station or near a cell tower running on a specialised hardware. The core is monolithic, centralise and far from devices and the eNodeB. This has been overcome in 5G, as mentioned before, the core has been disaggregated and each function can run in an independent way on off-the-shelf hardware without the need of having dedicated and difficult to change hardware. In this way, 5G Core functions can be co-located with the applications in an edge datacentre enabling short path communications thus improving the speed and latency. What we have in this scenario are small data centres positioned at the edge of the network, close to the cell towers. This allows to not only have a low latency but also a high bandwidth. Moreover, with this approach it is possible to have the previously mentioned network slicing. One slide may be dedicated to high bandwidth, one to low latency and another one for massive IoT devices [11].

3.3 5G Core Structure

The mobile core network is responsible for many differentiated functions such as session management, mobility authentication and security. All three aspects of crucial importance when delivering a service [10]. All the network generations, including 5G at first, used a point-to-point (P2P) architecture but with the transition to cloud infrastructure and the need for diverse services have made the P2P architecture not the best option anymore. P2P architectures contain a large number of unique interfaces between functional elements, and each is connected to multiple adjacent elements creating a complex connections map resulting in dependencies between functions. These dependencies make it difficult to change a deployed architecture but 5G needs a more dynamic and agile architecture like a Service Based Architecture (SBA).

SBA performs the decoupling mentioned previously in this document, it separates the end-user service from the underlaying network and platform infrastructure. The 5G Core architecture is designed to be cloud-native, it should make use of Network Function Virtualisation and Software Defined Networking and use service-based interactions between control plane functions. SBA aligns very well with a microservices view of network function composition.

5G System is made of three main components: 5G Core, Radio Access Network (RAN) and User Equipment (UE). The below image gives a representation of the 5G system:

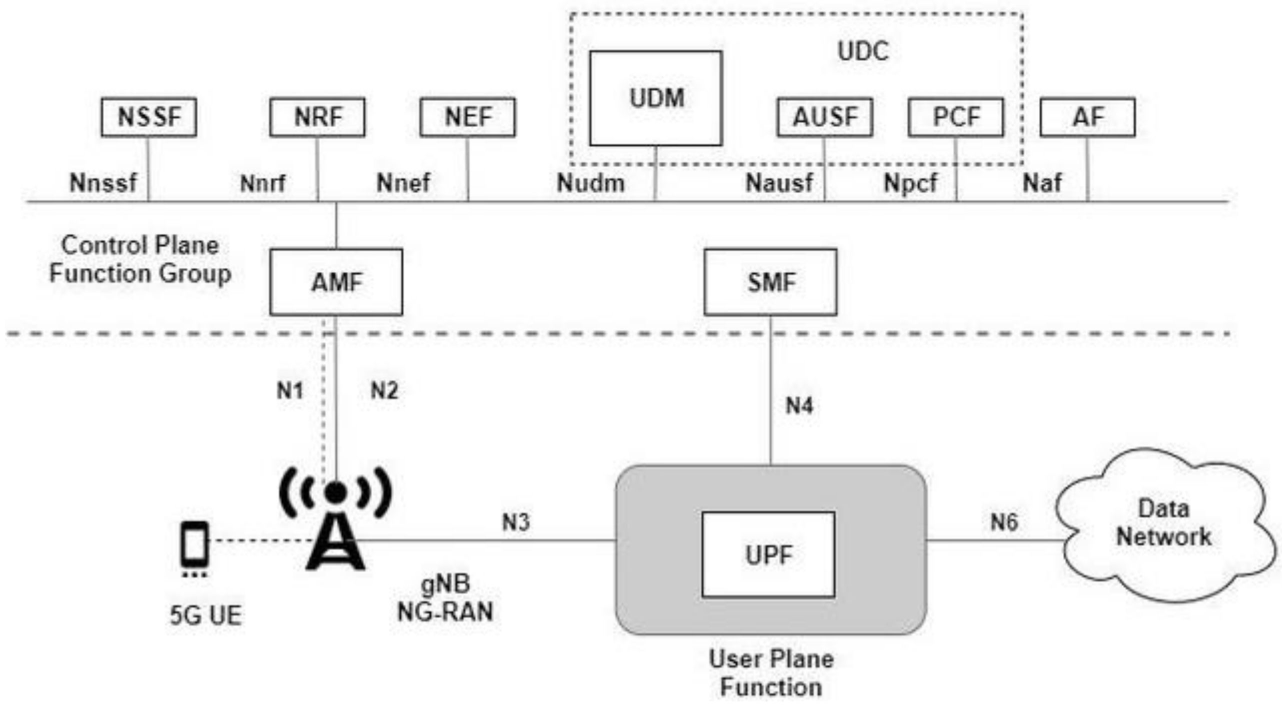


Figure 18 - 5G System

- ***User Equipment (UE) -*** like 5G smartphones or 5G cellular devices connect over the 5G New Radio Access Network to the 5G core and further to Data Networks (DN), like the Internet.
- ***gNB*** – a radio node that allows the UE to connect with the Core using the air interface. The gNB provides the user plane and control plane terminations towards UE.
- ***Access and Mobility Management Function (AMF)*** – manages access control and mobility. It also includes Network Slice Selection Function (NSSF).
- ***Session Management Function (SMF)*** – sets up and manages sessions according to network policy.
- ***User Plane Function (UPF)*** – it can be deployed in various configurations and locations according to the service type.
- ***Policy Control Function (PCF)*** – provides a policy framework incorporating network slicing, roaming and mobility management.
- ***Unified Data Management (UDM)*** – stores subscriber data and profiles.
- ***NF Repository Function (NRF) –*** provides registration and discovery functionality so that Network Functions (NFs) can discover each other and communicate via APIs.

- ***Network Exposure Function (NEF)*** **–** an API gateway that allows external users, such as enterprises or partner operators, the ability to monitor, provision and enforce application policy for user inside the operator network.
- ***Authentication Server Function (AUSF)*** **–** as the name implies, this is an authentication server.

Now that the an overall introduction to 5G has been provided along with details about the structure of the system and the core, the next chapter discusses the open source implementations of the technology.

3.4 5G Core Implementations

This chapter provides an overview of the three main open-source implementations of 5G Core: Open5GS, Free5GC and Magma. Then it attempts a comparison between these three platforms in terms of architecture, implementation and service features offered by each of them.

3.4.1 Open5GS

The Open5GS platform consists of two core architecture segments – the 4G/5G non-standalone core and 5G standalone core. The below image shows both options supported by Open5GS [13]:

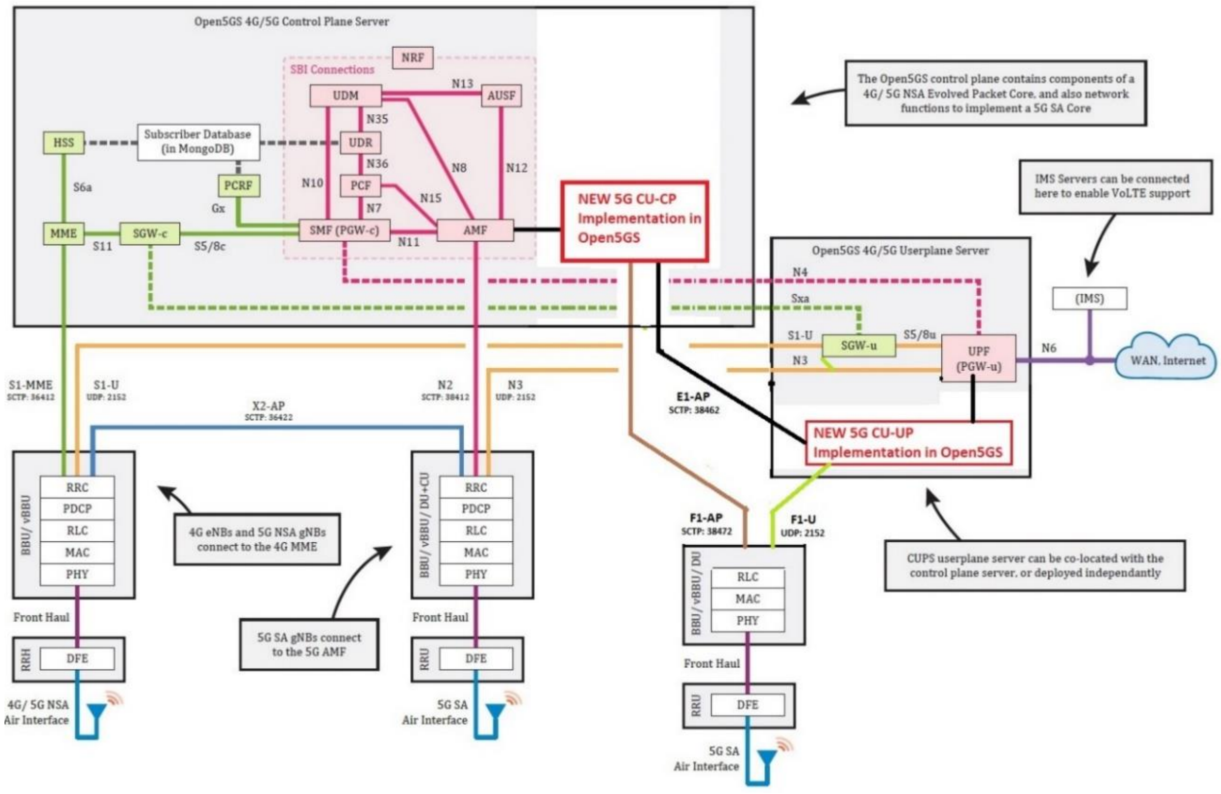

Open5GS 4G/5G Control Plane Server
SBI Connections
NRF
UDM
AUSF
N13
N35
UDR
N36
PCF
N8
N12
N10
N15
N7
HSS
Subscriber Database (in MongoDB)
PCRF
S6a
Gx
MME
SGW-c
SMF (PGW-c)
AMF
S11
S5/8c
N11
NEW 5G CU-CP Implementation in Open5GS
The Open5GS control plane contains components of a 4G/5G NSA Evolved Packet Core, and also network functions to implement a 5G SA Core
IMS Servers can be connected here to enable VoLTE support
Open5GS 4G/5G Userplane Server
N4
Sxa
S1-U
SGW-u
S5/8u
N3
UPF (PGW-u)
N6
(IMS)
WAN, Internet
NEW 5G CU-UP Implementation in Open5GS
S1-MME SCTP: 36412
S1-U UDP: 2152
X2-AP SCTP: 36422
N2 SCTP: 38412
N3 UDP: 2152
E1-AP SCTP: 38462
BBU / vBBU
RRC
PDCP
RLC
MAC
PHY
4G eNBs and 5G NSA gNBs connect to the 4G MME
BBU / vBBU / DU+CU
Front Haul
RRH
DFE
5G SA gNBs connect to the 5G AMF
RRU
4G / 5G NSA Air Interface
5G SA Air Interface
F1-AP SCTP: 38472
F1-U UDP: 2152
BBU / vBBU / DU
CUPS userplane server can be co-located with the control plane server, or deployed independantly

Figure 19 - Open5GS Architecture

Open5GS 4G/5G NSA contains the following components: Mobility Management Entity - MME; Home Subscriber Server - HSS; Policy and Charging Rules Function - PCRF; Serving Gateway Control Plane - SGWC; Serving Gateway User Plane - SGWU; Gateway Control Plane - PGWC; and Gateway User Plane - PGWU. In this architecture, the core is physically divided into two main levels - the control level and the user level. The MME is the main component of the control plane axis in the core. It is responsible for managing mobility, paging, and carrier sessions. The MME connects to the HSS, which generates SIM authentication vectors and manages the subscriber profile. It also connects to the gateway server control planes - SGWC and PGWC/SMF. All eNBs of the mobile network - 4G base stations - are connected to the MME. The PCRF is the last component of the control plane. It works between the PGWC/SMF and the HSS to calculate and enforce subscriber policies.

The user plane is responsible for transporting user data packets between the eNB/NSA gNB - 5G NSA base station - and the external WAN. SGWU and PGWU/UPF are the main components of the user plane and both are connected to their control plane counterparts, i.e., eNBs/NSA gNBs connect to SGWU, which in turn connects to PGWU/UPF and WAN. All these components have configuration files that contain IP/local interface names of the component and IP addresses/DNS names of other components to which each component must connect.

Open5GS 5G SA Core includes the following features: Access and Mobility Management Function - AMF; Session Management Function - SMF; User Plane Function - UPF; Authentication Server Function - AUSF; Repository Function - NRF; Unified Data Management - UDM; Unified Data Repository - UDR; Policy and Charging Function - PCF; Network Slice Selection Function - NSSF; and Binding Support Function - BSF [12]. In the control plane, the functions are configured to be registered in NRF. NRF helps in detecting other core functions. AMF function acts as mobility and connectivity manager. The gNBs - 5G base stations - connect to the AMF. UDM, AUSF and UDR perform similar operations as 4G HSS, generate SIM authentication vectors and maintain subscriber profile. The SMF is responsible for session management, which was previously the responsibility of the 4G MME/SGWC/PGWC. The NSSF function provides a way to select parts of the network. Ultimately, PCF is responsible for charging and subscriber policy enforcement. In the 5G SA core, only one user plane function is used, which makes it simpler compared to the 4G/5G NSA core. In it, the UPF function connects to the SMF and also transports user data packets between the gNB and the external WAN. Except for SMF and UPF, all configuration files for the 5G SA main functions contain only the IP binding addresses/local interface names of the function and the IP address/DNS names of the NRF [13].

3.4.2 Free5GC

Free5GC is an open-source project for the 5th generation mobile network core - 5G. Currently, the largest contributions to the development of the Free5GC platform come from National Chiao Tung University - NCTU. The Free5GC implementation is based on NextEPC. Since the commercial 5G UE and 5G base station - gNB - are not yet on the market, Free5GC uses 4G protocols to communicate with the 4G UE and 4G base station - eNB [14].

So, the authentication protocol of this platform is still based on 4G. Currently, Free5GC already implements some of the 5GC features required for its use. The image below illustrates the changes made to the core architecture of the network [14]:

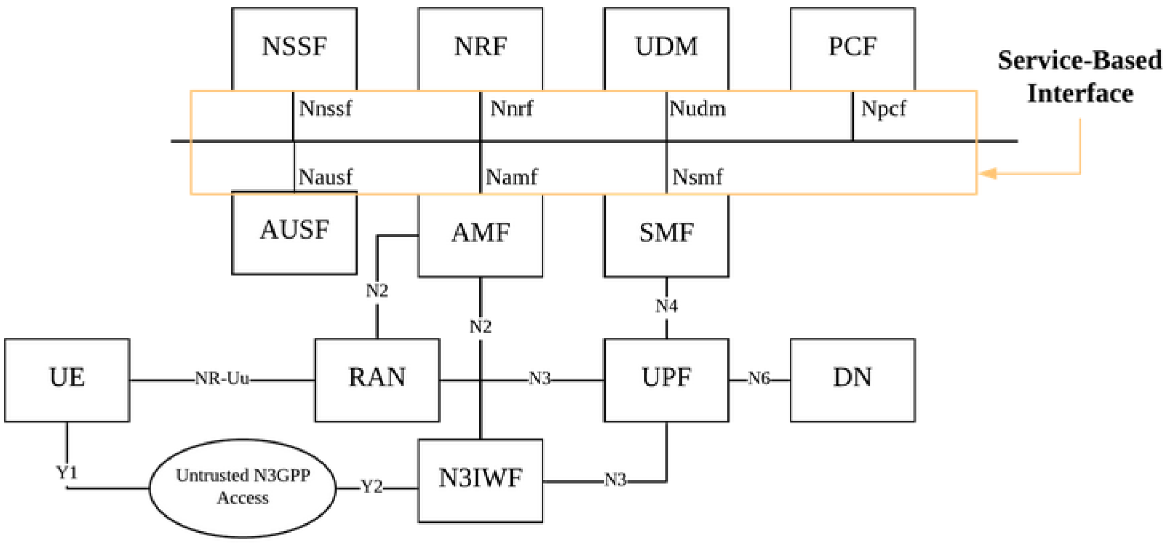


Figure 20 - Free5GC Architecture

Here the structure is more simple compared to the Open5GS but is also less complete in terms of elements of the core and the scenarios that can be covered by this implementation.

3.4.3 Magma Core

Magma is an open-source software platform that gives network operators an open, flexible and extendable mobile core network solution. The figure below shows the high-level architecture of Magma [14]:

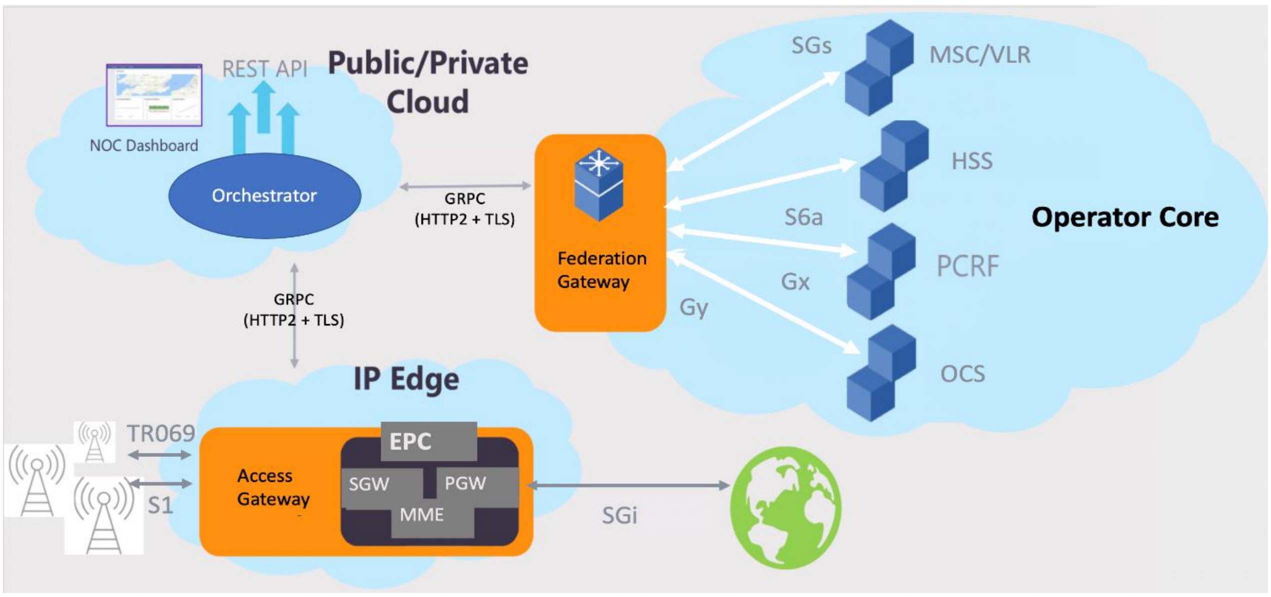


Figure 21 - Magma architecture

It has three main components:

- Access Gateway: The Access Gateway (AGW) provides network services and policy enforcement. In an LTE network, the AGW implements an evolved packet core (EPC), and a combination of an AAA and a PGW. It works with existing, unmodified commercial radio hardware.
- Orchestrator: Orchestrator is a cloud service that provides a simple and consistent way to configure and monitor the wireless network securely. The Orchestrator can be hosted on a public/private cloud. The metrics acquired through the platform allows you to see the analytics and traffic flows of the wireless users through the Magma web UI.
- Federation Gateway: The Federation Gateway integrates the MNO core network with Magma by using standard 3GPP interfaces to existing MNO components. It acts as a proxy between the Magma AGW and the operator's network and facilitates core functions, such as authentication, data plans, policy enforcement, and charging to stay uniform between an existing MNO network and the expanded network with Magma.

Magma provides network operators with an open, flexible and extensible packet core for mobile networks. It also enables better connectivity by allowing operators to offer mobile services independently of core network providers. It enables more efficient network management with automated systems, with less downtime, and improves predictability and agility for adding new features. It is also possible to expand the licensed spectrum around constrained operators as it allows the expansion of capacity and coverage through Wi-Fi and Citizens Broadband Radio Service (CBRS). Finally, it allows interconnection between existing Mobile Network Operators - MNOs and new infrastructure providers, resulting in the expansion of rural infrastructure. Magma provides a single panel for managing Magma-based networks. It provides services for configuring gateways and eNodeBs, as well as an overview of status, events and metrics for analysing the performance of the network and the devices connected to it. It is also possible to configure and receive alerts for failures, overloads, etc. [14].

3.4.4 Comparison Between 5G Core Implementations

A comparison between these three implementations allowed to understand which one was the most suitable one for this thesis work. For all three implementations it is important to consider different aspects such as infrastructure issues, platform documentation, developer community engagement and code maturity. Functional aspects such as mobility, scalability, reliability and latency are important too.

Another important aspect to consider is the underlaying hardware needed to support these implementations. The elements of all three components may differ in terms of the equipment needed to deploy them. There can be elements deployed over containers or over Virtual Machines. The below table shows how components are deployed:

Open5GS	**Free5GC**	**Magma**
Containers (Docker)	Virtual Machine	Container (Docker) + Bare Metal

Documentation is another important factor when deploying a 5G core infrastructure, the amount of documentation made available by the platforms and community is crucial to have an easy and efficient provisioning of the components. Open5GS and Magma provide a good quantity of instructions to perform the deployment, configuration and upgrade in a well-defined manner. In Free5GC the documentation has the disadvantage of not showing a chronological order of the execution process making the whole process of deployment and configuration a non-trivial task. Another important factor is how mature the code is and how active the community is. All three platforms have an active community which helps to improve the code, develop new features and fix bugs. The projects of all of them are still under development but Open5GS and Magma are more advanced compared to Free5GC.

When it comes to the control and user plane separation, both Open5GS and Free5GC provide a complete separation between these two planes making the functions independent. Magma does not support this separation yet. Its architecture consists of a control plane and user plane that operate together at Access Gateway.

As a consequence of these aspects, Open5GS has been selected for this thesis work as it allows to have a clean separation between the control and user plane, its deployment can be performed in a cloud-native way and has an active community of contributors to the project along with a well defined and rich documentation.

4. 5G System Deployment with Kubernetes

Applying a cloud native approach to 5G is essential to benefit from its structure and achieve the flexibility needed in different scenarios like the ones already mentioned in the previous chapter. Being cloud native implies that the obtained solution is scalable, portable and can be dynamically managed and orchestrated. However, being cloud native when working with telecommunication systems is challenging as they have very specific requirements that haven’t been fulfilled and tested yet or sometimes, in traditional orchestration frameworks, are not possible to satisfy. While recently major MANO frameworks claim that they have become cloud native, in reality this amounts to containerizing the MANO engines and executing them on top of container orchestrators, such as K8s. The workflow orientation, centralization of the management logic, and limited support for highly dynamic workloads still permeate the MANO frameworks. This is the reason behind a proposal of a new solution based entirely on Kubernetes [16].

This chapter provides an overview of the proposed solution and then it shifts the focus on the implementation part giving details about the Kubernetes cluster, deployment of 5G Core and UERANSIM.

4.1 Solution Architecture

What we are proposing in this thesis work is to move towards other technologies that allow to be cloud native in the true sense of the concept. MANO is a standard so it can be implemented in a multitude of ways, the idea is to use Kubernetes as the implementation of the management and orchestration module defined in the ETSI MANO standard. Rather than having a deployment of frameworks like OSM as entities running inside containers and acting as orchestrator for VNFs, here the proposed solution is to have Kubernetes acting as not just a Network Function Virtualisation Infrastructure (NFVI) but also as Network Function Virtualisation Orchestrator (NFVO), VNF Manager and VIM. This means having Kubernetes as a cloud native implementation opposed to the existing MANO orchestration frameworks. In this way it is possible to have a more portable and streamlined solution compared to having separate orchestration engines on top of Kubernetes. The image below shows the solution architecture:

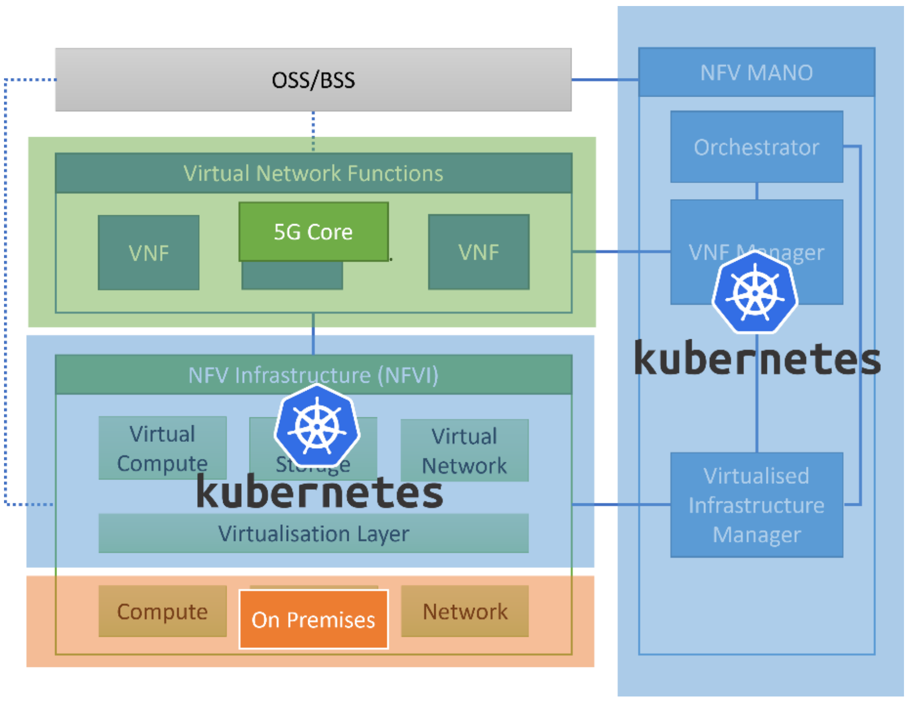


Figure 22 - Solution Infrastructure

As shown in the image, Kubernetes acts as an implementation of the MANO standard. So, it covers the main three components on the right side. The whole solution is hosted on a on-premises server with Ubuntu 18.04.6 LTS operating system, kernel Linux 5.4.0-117-generic and x86-64 architecture.

As discussed in chapter 3, the core will be made of different components representing VNFs and interacting one with another exploiting the inter-cluster networking. When it comes to the lifecycle of the VNFs, Kubernetes will be responsible for it. It will manage the creation of Pods, allocation of VNFs inside the Pods and ensure that they are up and running. The number of instances of each function can be decided also a priori with specification files as explained in chapter 2, while the load balancing will be directly performed by K8s.

4.2 Cluster Deployment

There are different technologies that can be used to create a Kubernetes cluster, and all differ in terms of the features they provide and scenarios they are more suitable for. Some can be mainly used to define a cluster from command line using configuration files while some others provide a very intuitive user interface to manage the cluster. The main ones are Kind, Minikube, Rancher and K3S. In this thesis work we have explored the first three to compare the differences between them and understand which one can be the best option for this project.

The next section compares Kind, Minikube and Rancher and then the chapter will focus on the deployment of the cluster.

4.2.1 Comparison Between Kind, Minikube and Rancher

Kind runs local Kubernetes clusters inside Docker container nodes, so the cluster is inside Docker containers which means that it has a fast start up speed compared to having a solution that is based on VMs. It provides a quite easy and straightforward way to create a cluster if the requirements are standard. Once kind has been installed on the machine, it is possible to create a cluster by simply running the *"kind create cluster"* command. At the same time, if there is a need of specific role in the cluster or a cluster with multiple nodes for example, it is possible to use configuration files to start the deployment. In case there is the necessity to define some more options when deploying the cluster, it is possible to use a configuration like the following:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
```

This allows to create a cluster that uses a specific apiVersion and creates two nodes: one control plane and another worker node. This file would allow to extend the cluster and add more worker nodes for example.

Usually when loading the container images, they have to be pushed to the registry and then they can be used and imported, but Kind allows to skip this step and host local images directly into the cluster. This functionality makes testing and development much quicker. Kind was in fact created to test Kubernetes but it can also be used for local development. When working with it and deploying the 5G core on top of a kind-created cluster, we have experienced issues related to a non-trivial configuration of the cluster. In order to achieve a successful deployment and a correctly running core from a functionality perspective, the cluster needed very verbose and manual configuration which can easily lead to errors due to cluster and network misconfigurations.

On the other hand, Minikube uses another approach based on exploiting a VM that is essentially a single node Kubernetes cluster. It provides support for all K8s features and a simple dashboard that can be used to have an overview of the cluster. Minikube provides a command to launch from the command line that allows to have a simple cluster made of a master node and a worker node: *minikube start.*

In both deployments, kubectl can be used to interact with the cluster be it for check its state or make changes to its structure. There is no need to perform extra steps, if the container images are ready to be used, it is possible to start the application using a YAML file and running the following command:

kubectl apply -f <YAML file>

The apply command manages application through files defining Kubernetes resources. When the command runs, it crates and updates resources in the cluster.

Minikube represents a tool which is more complete and easier to use compared to Kind, but it still has some limitations as again the creation and configuration of the cluster require some manual configuration. It still is a good option to be used compared to kind but for several reasons like ease of configuration in complex scenarios, completeness and variety of tools to deploy, manage and monitor applications an even better option is Rancher.

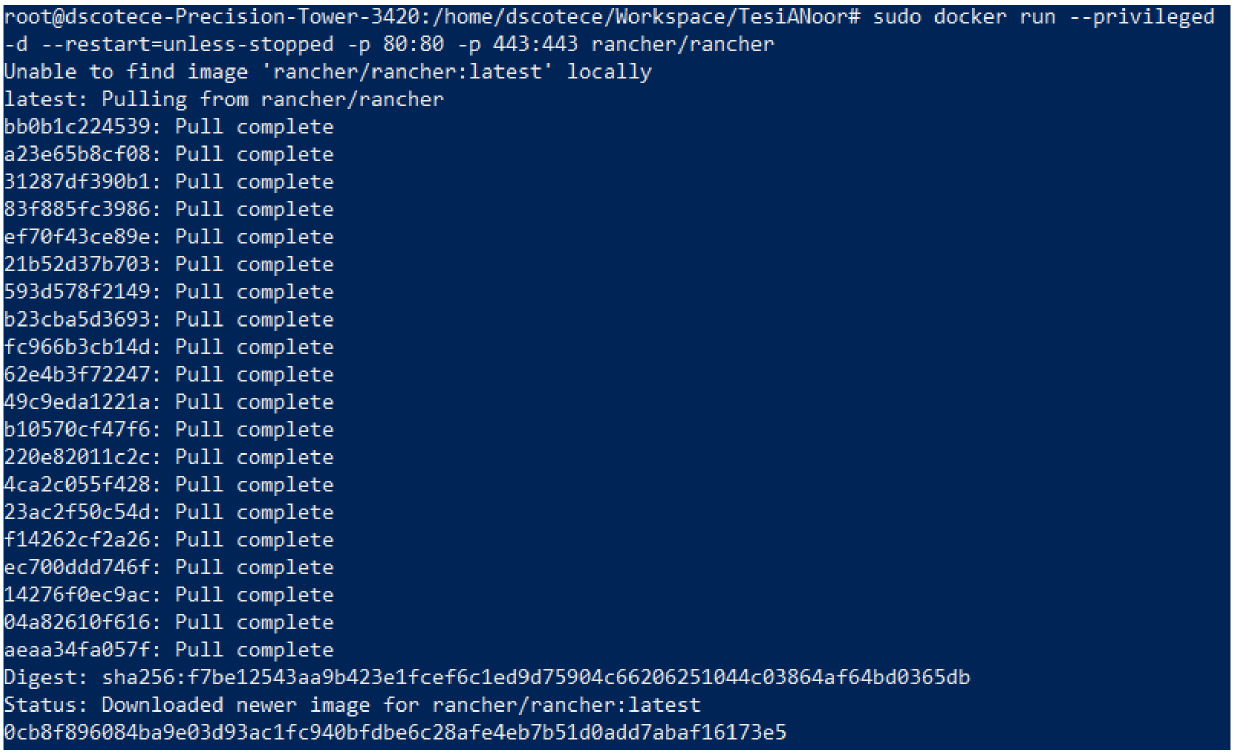
Rancher is one of the main tools for open-source cloud native Kubernetes management and allows to manage and scale workload in a consistent way in any environment [17]. The initial setup of a cluster is a little more complex compared to Kind and Minikube, but it provides a more complete cluster in terms of network configuration, overall management of the cluster and very well-defined and intuitive user interface to interact with the it. It has a variety of tools to include other platforms such as Helm for deployments or monitoring tools bases on Prometheus just to mention some of the options. It is possible to interact with the cluster via kubectl but also from the UI which makes the admin work much easier. The next chapter explains how it is possible to create a cluster in Rancher and the features needed for the project.

4.2.2 Cluster with Rancher

To get started with Rancher and be able to create the cluster there are two preliminary steps to be performed: have a Linux host and a supported version of Docker running on it. In our case the host is a on-premises machine with the previously mentioned features. The docker version is 20.10.16. To install and run Rancher it is necessary to run the following command:

```
$ sudo docker run --privileged -d --restart=unless-stopped -p 80:80 -p 443:443 rancher/rancher
```

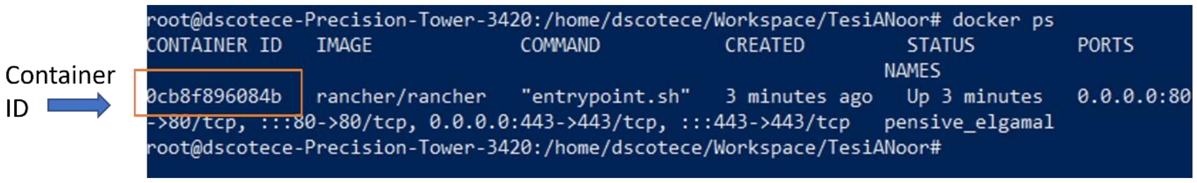
This command will result in the download of a set of images that are needed to run Rancher:



```
root@dscotece-Precision-Tower-3420:/home/dscotece/Workspace/TesiANoor# sudo docker run --privileged
-d --restart=unless-stopped -p 80:80 -p 443:443 rancher/rancher
Unable to find image 'rancher/rancher:latest' locally
latest: Pulling from rancher/rancher
bb0b1c224539: Pull complete
a23e65b8cf08: Pull complete
31287df390b1: Pull complete
83f885fc3986: Pull complete
ef70f43ce89e: Pull complete
21b52d37b703: Pull complete
593d578f2149: Pull complete
b23cba5d3693: Pull complete
fc966b3cb14d: Pull complete
62e4b3f72247: Pull complete
49c9eda1221a: Pull complete
b10570cf47f6: Pull complete
220e82011c2c: Pull complete
4ca2c055f428: Pull complete
23ac2f50c54d: Pull complete
f14262cf2a26: Pull complete
ec700ddd746f: Pull complete
14276f0ec9ac: Pull complete
04a82610f616: Pull complete
aeaa34fa057f: Pull complete
Digest: sha256:f7be12543aa9b423e1fcef6c1ed9d75904c66206251044c03864af64bd0365db
Status: Downloaded newer image for rancher/rancher:latest
0cb8f896084ba9e03d93ac1fc940bfdbe6c28afe4eb7b51d0add7abaf16173e5
```

Figure 23 - Cluster Creation

This will create the following container on the hosting machine:



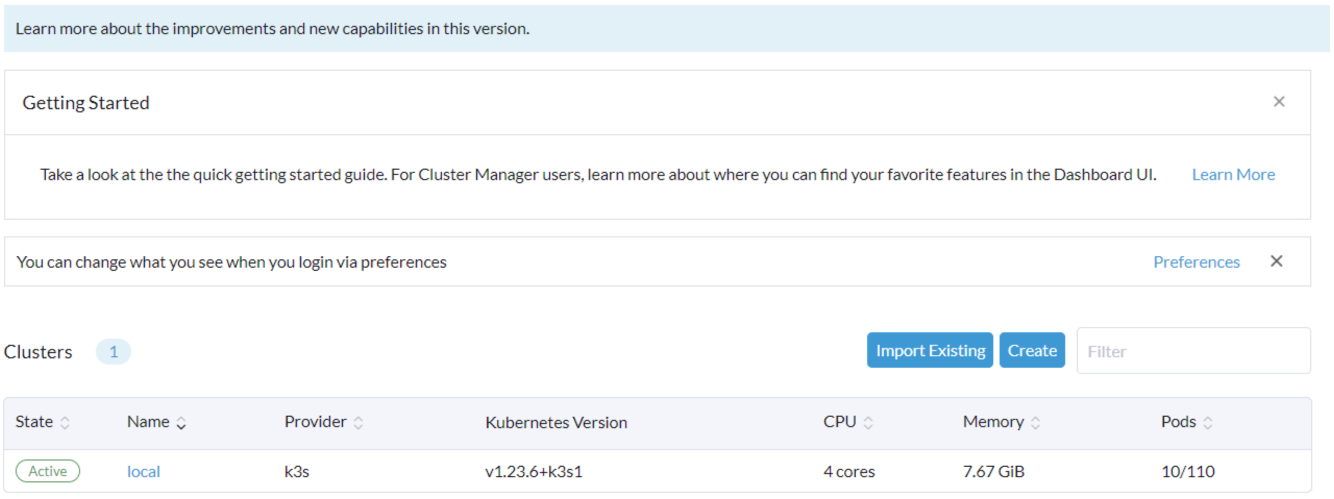
```
root@dscotece-Precision-Tower-3420:/home/dscotece/Workspace/TesiANoor# docker ps
CONTAINER ID   IMAGE             COMMAND           CREATED         STATUS         PORTS
                                                                   NAMES
0cb8f896084b   rancher/rancher   "entrypoint.sh"   3 minutes ago   Up 3 minutes   0.0.0.0:80
->80/tcp, :::80->80/tcp, 0.0.0.0:443->443/tcp, :::443->443/tcp   pensive_elgamal
root@dscotece-Precision-Tower-3420:/home/dscotece/Workspace/TesiANoor#
```

Figure 24 - Docker Container Hosting Rancher

It is now possible to access the Rancher Server UI by opening up a browser and connecting to the IP address or hostname where the command was launched. We now need to create a password to access Rancher from the UI, and to do so we need to run the following command:

docker logs <container-id> 2>&1 | grep "Bootstrap Password:"

This allows to get the password from the hosting machine and then add it on Rancher UI. Once this step has been completed the user can start to create a cluster with the desired features. As shown in the below image, in the list of existing clusters there is a cluster called "local" which is hosting Rancher and should not be used for other purposes. Usually, only admins have access to that cluster and other users should be prevented from accessing it as some permissions on that cluster may give them unnecessary privileged access to other cluster as well.


Learn more about the improvements and new capabilities in this version.

Getting Started ×

Take a look at the the quick getting started guide. For Cluster Manager users, learn more about where you can find your favorite features in the Dashboard UI. Learn More

You can change what you see when you login via preferences Preferences ×

Clusters 1 Import Existing Create Filter

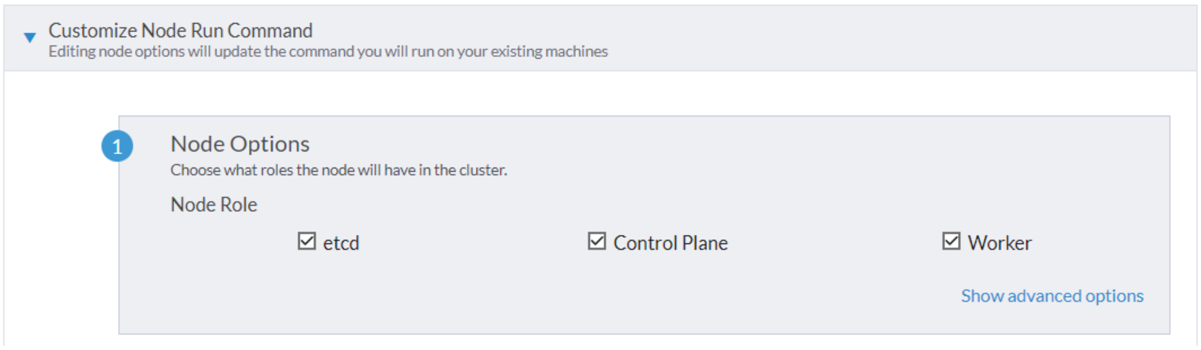
State	Name	Provider	Kubernetes Version	CPU	Memory	Pods
Active	local	k3s	v1.23.6+k3s1	4 cores	7.67 GiB	10/110

Figure 25- Clusters List

It is now possible to create a new cluster that will host the 5G System. For this purpose, many are the available choices, for example Amazon EKS, Azure AKS and Google GKE. But in our case, we have an on-premises server, so a “custom cluster” is what we need. As mentioned before, one of the best features of Rancher is that is allows to set-up different options for the cluster in the process of creating it. It allows to choose the version of Rancher that the user needs depending on the application requirements, it also allows to set-up a network provider that will allow to manage and change the network connectivity inside the cluster.

In this project it is necessary to make some changes to the network connectivity inside the cluster and to do so we need to disable the default configurations and enable a third-party plugin like ***calico.*** It makes the full network connectivity more flexible and avoids having the complexity of many YAML files configuring the network with NetworkPolicy API [18].

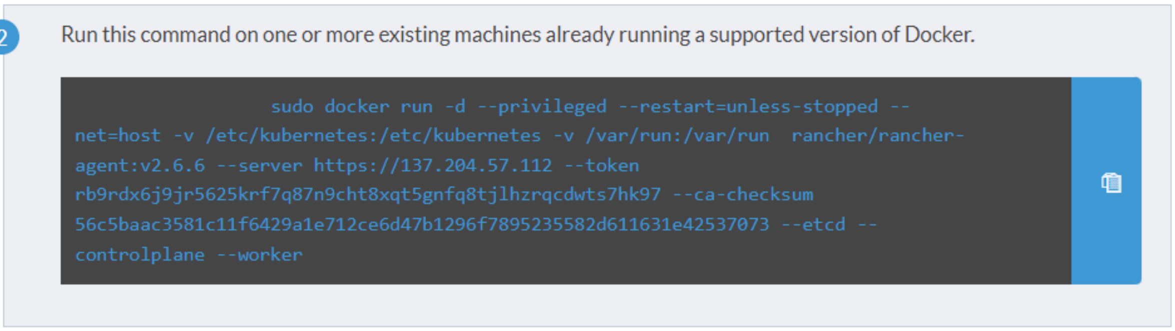
At the time of cluster creation, it is also possible to define the root directory for Docker, so we know on the server machine where all the data related to Docker is stored. Rancher also provides the possibility to select the roles of the node, in our case all three roles will be covered by the node:


Customize Node Run Command
Editing node options will update the command you will run on your existing machines

1 Node Options
Choose what roles the node will have in the cluster.
Node Role
☑ etcd ☑ Control Plane ☑ Worker

Figure 26 - Node Options in Cluster

Once the settings are completed, Rancher provided the following command to run from the command line so to start instantiating the cluster:


2 Run this command on one or more existing machines already running a supported version of Docker.

```
                sudo docker run -d --privileged --restart=unless-stopped --
net=host -v /etc/kubernetes:/etc/kubernetes -v /var/run:/var/run  rancher/rancher-
agent:v2.6.6 --server https://137.204.57.112 --token
rb9rdx6j9jr5625krf7q87n9cht8xqt5gnfq8tjlhzrqcdwts7hk97 --ca-checksum
56c5baac3581c11f6429a1e712ce6d47b1296f7895235582d611631e42537073 --etcd --
controlplane --worker
```

Figure 27 - Command to Instantiate the Cluster

This will then start the process of cluster creation and the following containers will be created on the host (the image shows only the main containers, the deployment has a set of other containers running as well):

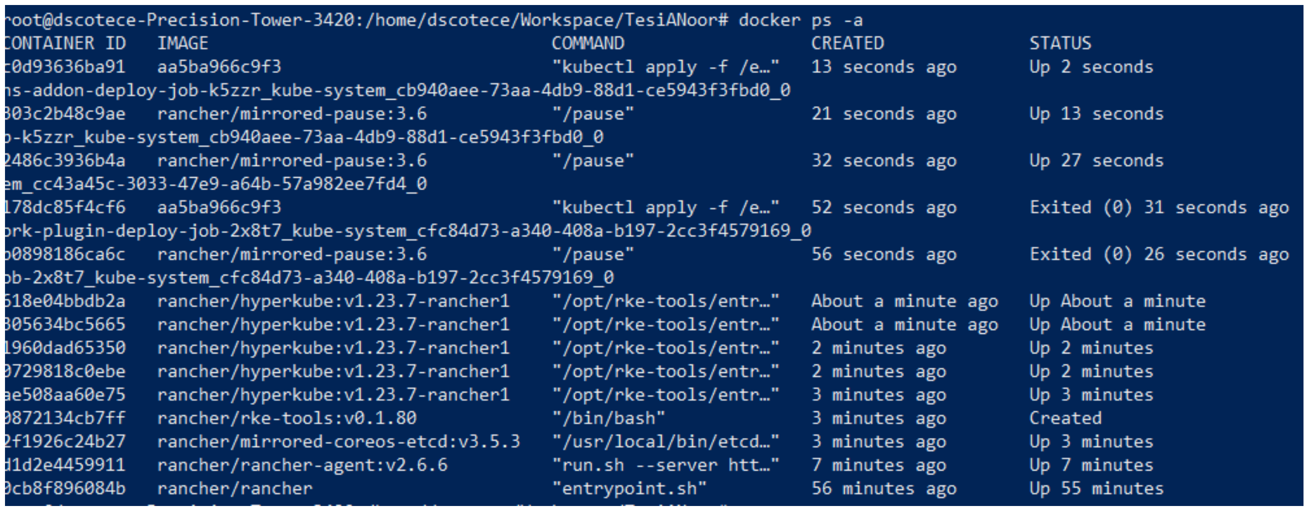
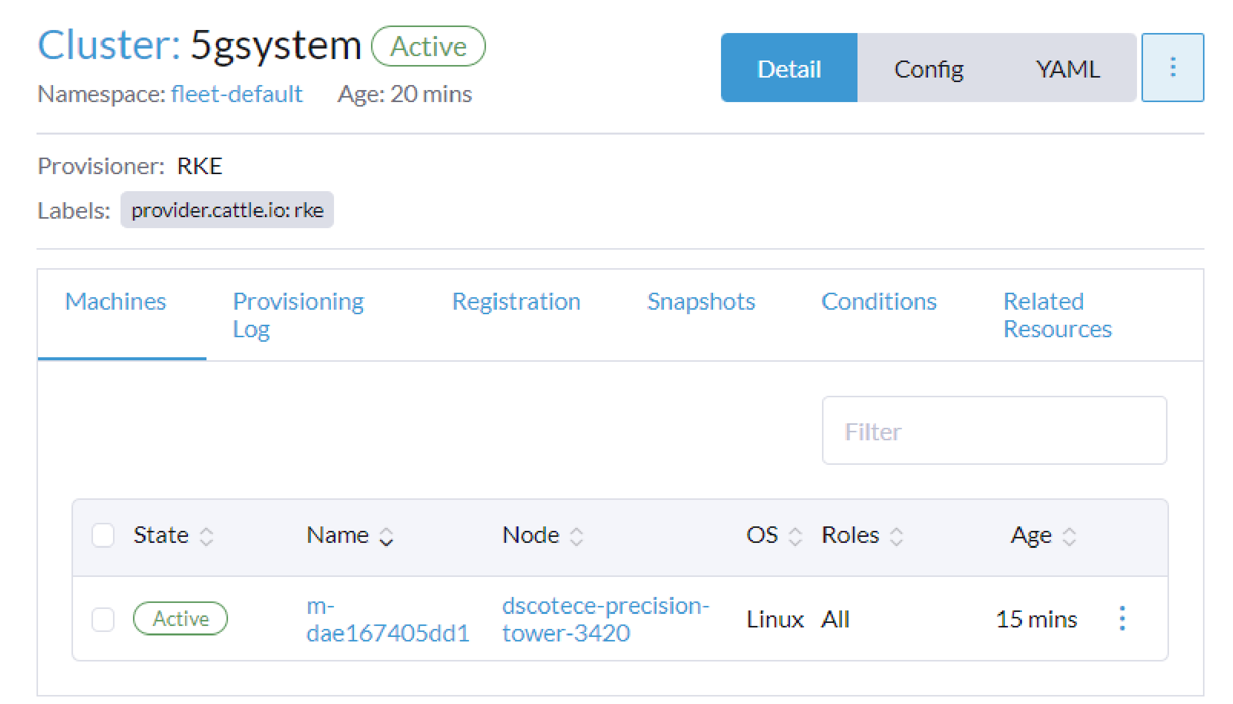

```
root@dscotece-Precision-Tower-3420:/home/dscotece/Workspace/TesiANoor# docker ps -a
CONTAINER ID   IMAGE                                COMMAND                  CREATED              STATUS
c0d93636ba91   aa5ba966c9f3                         "kubectl apply -f /e…"   13 seconds ago       Up 2 seconds
ns-addon-deploy-job-k5zzr_kube-system_cb940aee-73aa-4db9-88d1-ce5943f3fbd0_0
303c2b48c9ae   rancher/mirrored-pause:3.6           "/pause"                 21 seconds ago       Up 13 seconds
b-k5zzr_kube-system_cb940aee-73aa-4db9-88d1-ce5943f3fbd0_0
2486c3936b4a   rancher/mirrored-pause:3.6           "/pause"                 32 seconds ago       Up 27 seconds
em_cc43a45c-3033-47e9-a64b-57a982ee7fd4_0
178dc85f4cf6   aa5ba966c9f3                         "kubectl apply -f /e…"   52 seconds ago       Exited (0) 31 seconds ago
ork-plugin-deploy-job-2x8t7_kube-system_cfc84d73-a340-408a-b197-2cc3f4579169_0
b0898186ca6c   rancher/mirrored-pause:3.6           "/pause"                 56 seconds ago       Exited (0) 26 seconds ago
ob-2x8t7_kube-system_cfc84d73-a340-408a-b197-2cc3f4579169_0
518e04bbdb2a   rancher/hyperkube:v1.23.7-rancher1   "/opt/rke-tools/entr…"   About a minute ago   Up About a minute
305634bc5665   rancher/hyperkube:v1.23.7-rancher1   "/opt/rke-tools/entr…"   About a minute ago   Up About a minute
1960dad65350   rancher/hyperkube:v1.23.7-rancher1   "/opt/rke-tools/entr…"   2 minutes ago        Up 2 minutes
0729818c0ebe   rancher/hyperkube:v1.23.7-rancher1   "/opt/rke-tools/entr…"   2 minutes ago        Up 2 minutes
ae508aa60e75   rancher/hyperkube:v1.23.7-rancher1   "/opt/rke-tools/entr…"   3 minutes ago        Up 3 minutes
0872134cb7ff   rancher/rke-tools:v0.1.80            "/bin/bash"              3 minutes ago        Created
2f1926c24b27   rancher/mirrored-coreos-etcd:v3.5.3  "/usr/local/bin/etcd…"   3 minutes ago        Up 3 minutes
d1d2e4459911   rancher/rancher-agent:v2.6.6         "run.sh --server htt…"   7 minutes ago        Up 7 minutes
0cb8f896084b   rancher/rancher                      "entrypoint.sh"          56 minutes ago       Up 55 minutes
```

Figure 28 - Cluster Creation

On the User Interface it is possible to see the newly created cluster up and running:



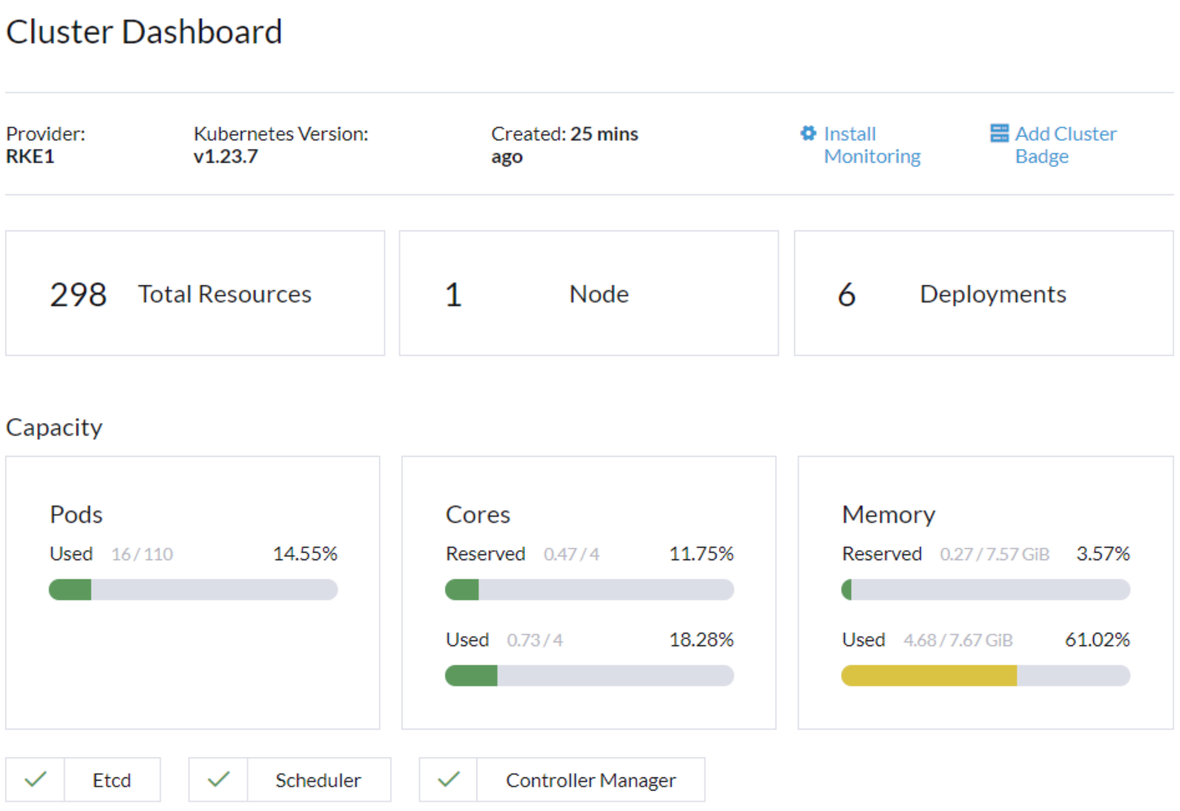
Cluster: 5gsystem Active
Namespace: fleet-default Age: 20 mins

Provisioner: RKE
Labels: provider.cattle.io: rke

State	Name	Node	OS	Roles	Age
Active	m-dae167405dd1	dscotece-precision-tower-3420	Linux	All	15 mins

Figure 29 - Cluster Created

The below image shows the features of the cluster needed for the deployment:



Cluster Dashboard

Provider: RKE1
Kubernetes Version: v1.23.7
Created: 25 mins ago

298 Total Resources
1 Node
6 Deployments

Capacity

Pods	Used	16 / 110	14.55%
Cores	Reserved	0.47 / 4	11.75%
Cores	Used	0.73 / 4	18.28%
Memory	Reserved	0.27 / 7.57 GiB	3.57%
Memory	Used	4.68 / 7.67 GiB	61.02%

✓ Etcd ✓ Scheduler ✓ Controller Manager

Figure 30 - Rancher Dashboard

4 Cores have been dedicated to the node along with more than 7GB of memory. The dashboard shows an overview of the resource consumption whenever a new deployment it performed on the cluster. This is very useful to have a “live” knowledge about the resources’ status.

As mentioned previously, Rancher allows to manage from the dashboard different aspects of the cluster. Managing the workload from the UI allows to manage Pods and Deployments in an efficient manner compared to deploying from command line. The below figure pictures the options provided on the UI to manage not only Workloads but also services, storage and apps:

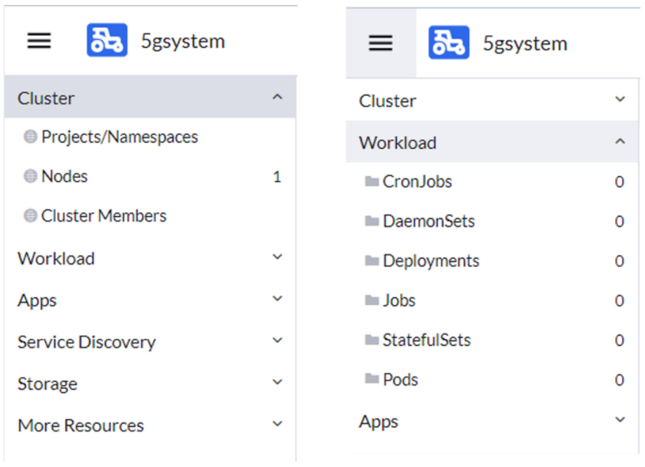
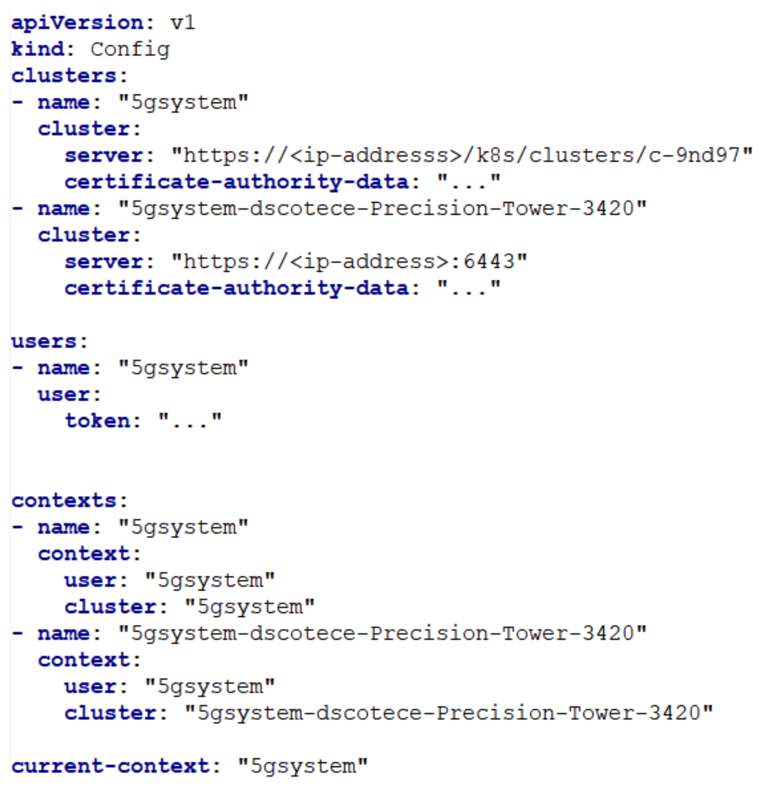


Figure 31 - Cluster Management Tools

Even if in some instances the UI is very useful, the command line stays the most powerful tool to communicate with the cluster and manage it. For this reason, in the following chapters there are actions performed using both tools. In order to enable kubectl to perform changes on the cluster, it is necessary to download the ***kubeConfig*** file from the cluster dashboard and update it on the host machine. Follows an example of a configuration file:



```yaml
apiVersion: v1
kind: Config
clusters:
- name: "5gsystem"
  cluster:
    server: "https://<ip-addresss>/k8s/clusters/c-9nd97"
    certificate-authority-data: "..."
- name: "5gsystem-dscotece-Precision-Tower-3420"
  cluster:
    server: "https://<ip-address>:6443"
    certificate-authority-data: "..."

users:
- name: "5gsystem"
  user:
    token: "..."


contexts:
- name: "5gsystem"
  context:
    user: "5gsystem"
    cluster: "5gsystem"
- name: "5gsystem-dscotece-Precision-Tower-3420"
  context:
    user: "5gsystem"
    cluster: "5gsystem-dscotece-Precision-Tower-3420"

current-context: "5gsystem"
```

Figure 32 - KubeConfig File

Some parts of the configuration have been replaced by "…" only for the sake of understanding better the format of the file. The file contains information about the cluster, the IP address on which it is hosted, the users and a token to allows the access from command line along with some contexts.

Now that the cluster has been successfully created, it is possible to start the 5G System deployment. The next chapters show how to successfully deploy the 5G Core and a UERANSIM so to have the entire system on a Kubernetes Cluster.

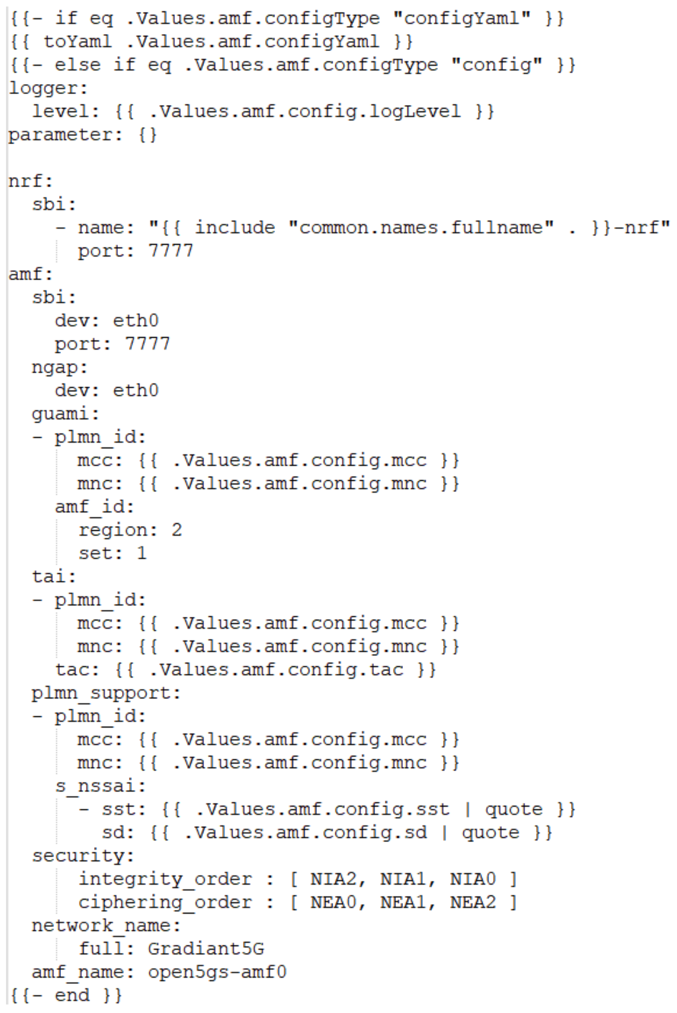
4.3 5G Core Deployment

The deployment of the 5G Core has a certain complexity as the application has many modules requiring very specific configurations, and this can very easily lead to a complicated scenario. To avoid problems due to complex and big quantity of configurations, helm has been used to manage the deployment.

Helm is a tool for managing Kubernetes packages called charts which are a collection of files that describe a related set of K8s resources [19]. Helm's only prerequisite is to have a Kubernetes cluster and charts needed to deploy the application. Once it has been installed, it is important to initialise the helm chart repository as this can be useful to build the dependencies when doing a deployment:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Once the dependencies have been built, it is possible to start the deployment of 5G Core. We have YAML files for all components of the core: AMF, BSF, NRF, PCF, SGWC, SMF, UDR, AUSF, HSS, MME, NSSF, PCRF, SGWU, UDM AND UPF. All these files contain the configuration of these components and references to dependant components. An example of these files is the amf.yaml one as shown in the image below:

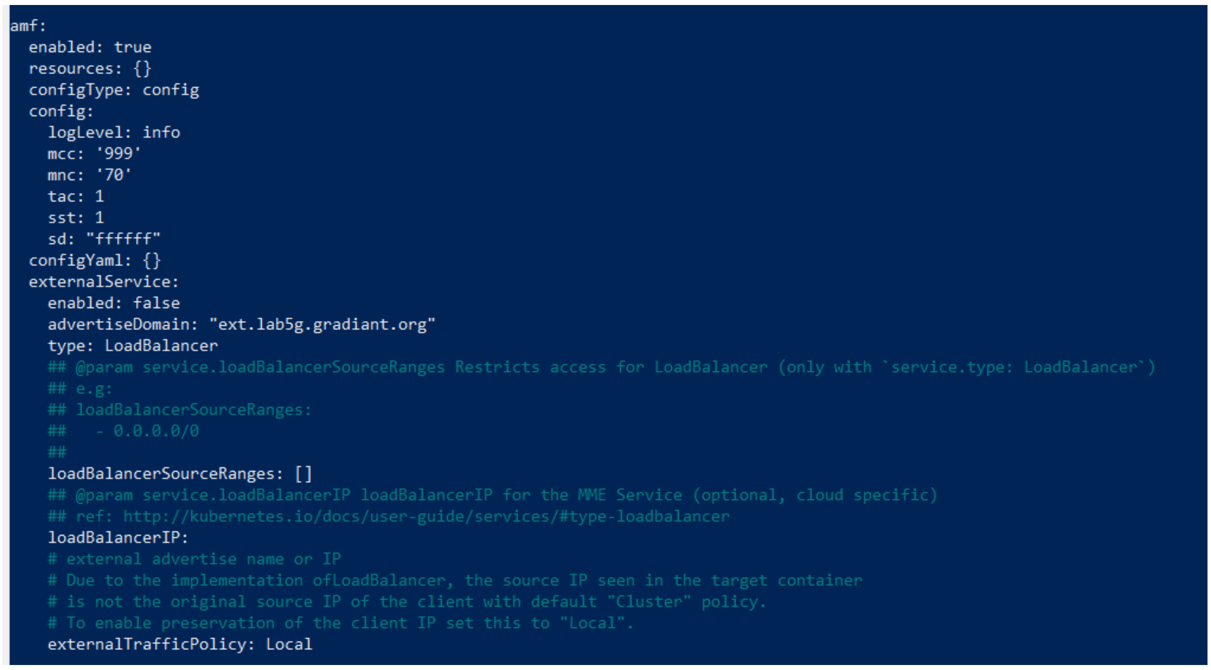


```
{{- if eq .Values.amf.configType "configYaml" }}
{{ toYaml .Values.amf.configYaml }}
{{- else if eq .Values.amf.configType "config" }}
logger:
  level: {{ .Values.amf.config.logLevel }}
parameter: {}

nrf:
  sbi:
    - name: "{{ include "common.names.fullname" . }}-nrf"
      port: 7777
amf:
  sbi:
    dev: eth0
    port: 7777
  ngap:
    dev: eth0
  guami:
  - plmn_id:
      mcc: {{ .Values.amf.config.mcc }}
      mnc: {{ .Values.amf.config.mnc }}
    amf_id:
      region: 2
      set: 1
  tai:
  - plmn_id:
      mcc: {{ .Values.amf.config.mcc }}
      mnc: {{ .Values.amf.config.mnc }}
    tac: {{ .Values.amf.config.tac }}
  plmn_support:
  - plmn_id:
      mcc: {{ .Values.amf.config.mcc }}
      mnc: {{ .Values.amf.config.mnc }}
    s_nssai:
      - sst: {{ .Values.amf.config.sst | quote }}
        sd: {{ .Values.amf.config.sd | quote }}
  security:
      integrity_order : [ NIA2, NIA1, NIA0 ]
      ciphering_order : [ NEA0, NEA1, NEA2 ]
  network_name:
      full: Gradiant5G
  amf_name: open5gs-amf0
{{- end }}
```

Figure 33 - example of AMF configuration file

This file contains many settings related to the hosting and communication inside the cluster. Information about the PLMN is added here and will enable the communication from UEs toward the network, forwarded by the gNodeB. Other network and security configurations are also set here. Values for the parameters are defined by some placeholders that will collect the information from an external configuration file. In fact, there is also a configuration file that acts as central point of configuration for all the component and will be used to set-up the whole core system. Using this file, it is possible to provide the settings for the deployment and make changes only to this file when an update is required, without the need to configure again each YAML file for the core components. The configuration file has a precise structure that must be kept to have a successful deployment and avoid errors, this is the reason why a schema file is necessary. It is a json file that contains the schema for the configurations and whenever there is a need for a specific change in the configurations, the schema file must be changes too in order to avoid any conversion problems of the parameters.

The most interesting configurations are for amf and mme components (they are very similar, so the image below reports only the amf ones):



```yaml
amf:
  enabled: true
  resources: {}
  configType: config
  config:
    logLevel: info
    mcc: '999'
    mnc: '70'
    tac: 1
    sst: 1
    sd: "ffffff"
  configYaml: {}
  externalService:
    enabled: false
    advertiseDomain: "ext.lab5g.gradiant.org"
    type: LoadBalancer
    ## @param service.loadBalancerSourceRanges Restricts access for LoadBalancer (only with `service.type: LoadBalancer`)
    ## e.g:
    ## loadBalancerSourceRanges:
    ##   - 0.0.0.0/0
    ##
    loadBalancerSourceRanges: []
    ## @param service.loadBalancerIP loadBalancerIP for the MME Service (optional, cloud specific)
    ## ref: http://kubernetes.io/docs/user-guide/services/#type-loadbalancer
    loadBalancerIP:
    # external advertise name or IP
    # Due to the implementation ofLoadBalancer, the source IP seen in the target container
    # is not the original source IP of the client with default "Cluster" policy.
    # To enable preservation of the client IP set this to "Local".
    externalTrafficPolicy: Local
```

Figure 34 - AMF configuration

Where the most important fields are [20]:

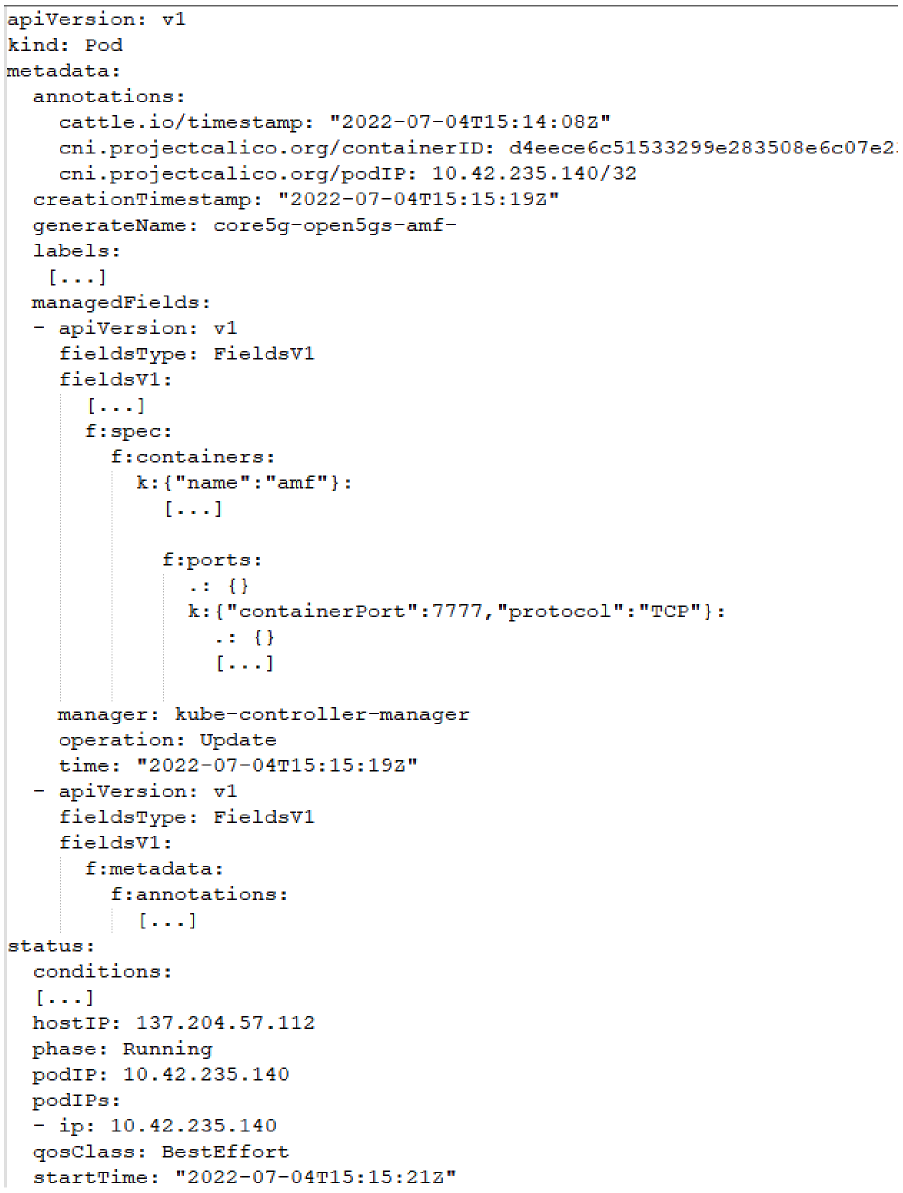
- **mcc**: Mobile Country Code value of HPLMN (Home PLMN).
- **mnc**: Mobile Network Code value of HPLMN (Home PLMN). It can be either 2 or 3 digits long.
- **tac**: Tracking Area Core value for the cell.

These parameters will be crucial when deploying the UERANSIM (explained later).

Once the parameter in the configuration file have been set, it is possible to start the deployment of the core with the following command:

```
helm install <deployment_name> <charts>
```

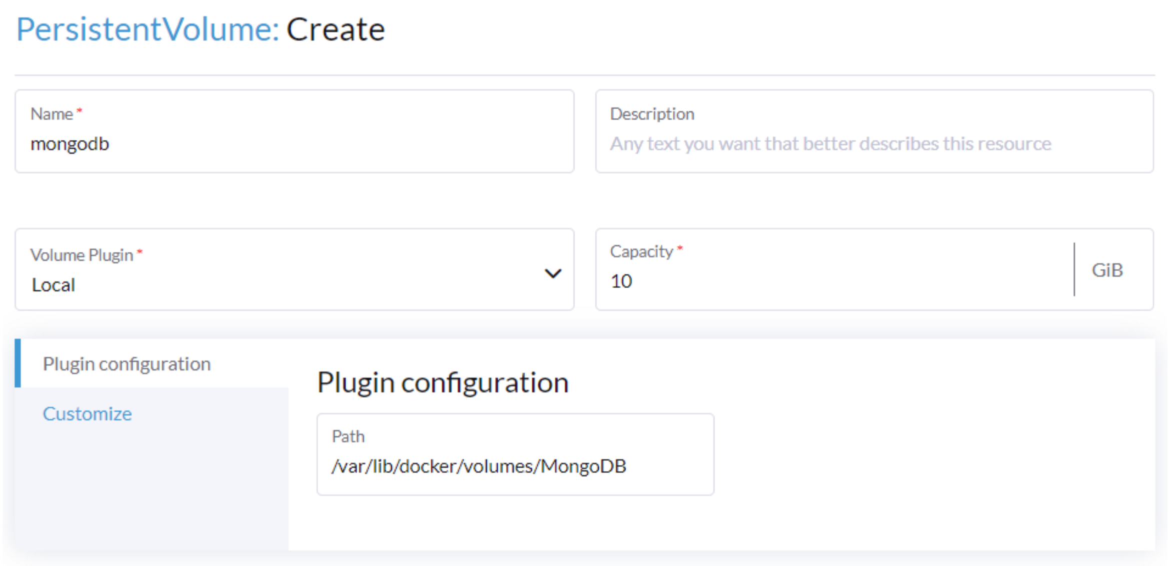
This will start a set of Pods on the cluster and each of them will be running one component of the core. Thanks to the configuration files mentioned above, YAML files for the Pods will be created and will allow to initialise the Pods and host the components inside them. An example of these file is provided in fig. 25, some parts of the file have been removed to make it more readable and focus on the main ones. Here we can find the ID of the container where the image is running, the IP address of the Pod, different fields and the name of the containers. It is interesting to notice that the ports that were defined in the configuration files mentioned above are listed also here and used to configure the ports of the container and the communication protocol. At the end of the file, we can find the status of the Pod. This is the same status mentioned in the chapter 2 where we explain how Kubernetes handles the scaling and components lifecycle depending on the comparison between the actual and desired state.



```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    cattle.io/timestamp: "2022-07-04T15:14:08Z"
    cni.projectcalico.org/containerID: d4eece6c51533299e283508e6c07e2
    cni.projectcalico.org/podIP: 10.42.235.140/32
  creationTimestamp: "2022-07-04T15:15:19Z"
  generateName: core5g-open5gs-amf-
  labels:
   [...]
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      [...]
      f:spec:
        f:containers:
          k:{"name":"amf"}:
            [...]

            f:ports:
              .: {}
              k:{"containerPort":7777,"protocol":"TCP"}:
                .: {}
                [...]

    manager: kube-controller-manager
    operation: Update
    time: "2022-07-04T15:15:19Z"
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          [...]
status:
  conditions:
  [...]
  hostIP: 137.204.57.112
  phase: Running
  podIP: 10.42.235.140
  podIPs:
  - ip: 10.42.235.140
  qosClass: BestEffort
  startTime: "2022-07-04T15:15:21Z"
```

Figure 35 - AMF Pod YAML

There are dependencies between the core elements to guarantee a correct deployment, i.e. Pods will wait for the database to be available before starting. The status and data from the cluster is saved into this database. As mentioned in the previous chapter, Kubernetes does not handle data persistency, so this must be done by the administrator. In the release there is a component called mongodb, it will act as the database for the cluster and at this point the concepts of PersistentVolume and PersistentVolumeClaim are very important. In fact, the database creates a PersistentVolumeClaim that requires to be bound to a PersistentVolume. This can be achieved either from the command line creating a configuration file for PV or using the Rancher Dashboard. The following images show how the PV is created in the latter case:


PersistentVolume: Create

Name *: mongodb

Description: Any text you want that better describes this resource

Volume Plugin *: Local

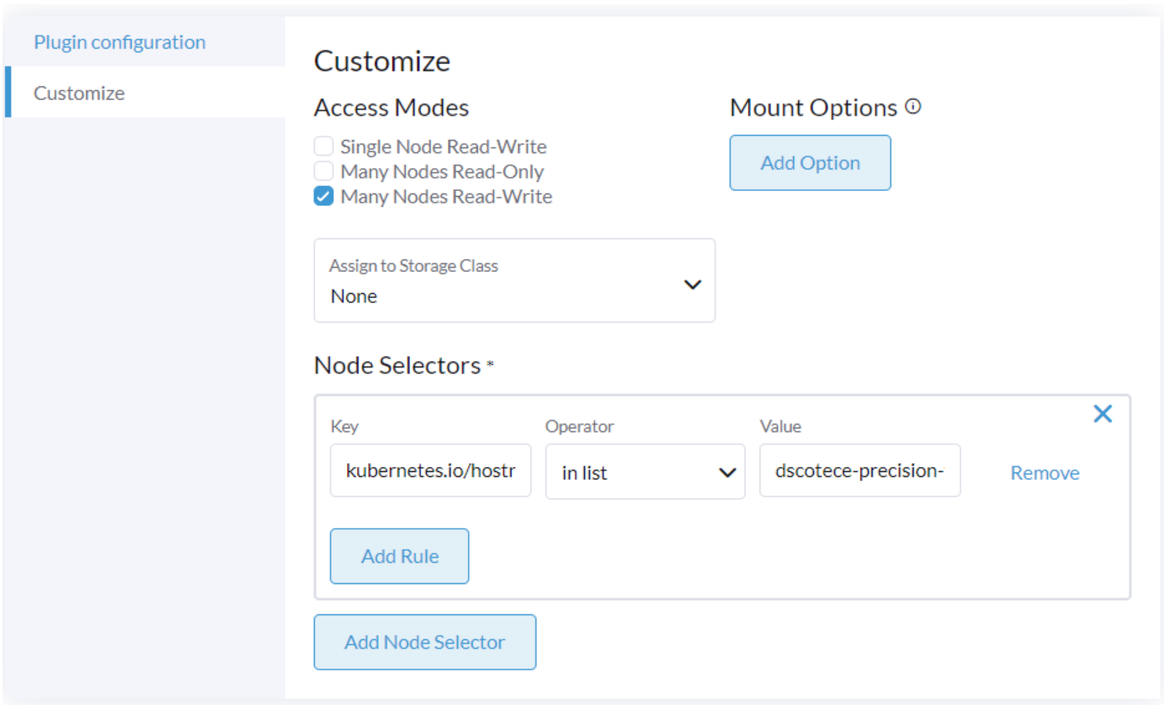
Capacity *: 10 GiB

Plugin configuration

Path: /var/lib/docker/volumes/MongoDB

Figure 36 - PersistentVolume Creation (a)

There must be affinity in the name of the claim and the volume, here it is called mongodb, so the claim will be able to understand which persistent volume has been created for it. There are different options when creating the volume, it can be hosted in local like in this case or on the cloud i.e. Amazon EBS Disk or Azure Disk. It is also possible to define the capacity of the volume and a path to the data.


Plugin configuration
Customize

Customize

Access Modes
- [ ] Single Node Read-Write
- [ ] Many Nodes Read-Only
- [x] Many Nodes Read-Write

Assign to Storage Class: None

Mount Options
Add Option

Node Selectors *

Key	Operator	Value	
kubernetes.io/hostr	in list	dscotece-precision-	Remove

Add Rule
Add Node Selector

Figure 37- PersistentVolume Creation (b)

In the customisation tab, there are different options to select from to have the necessary features. In this case the access mode has been set to "many nodes read-write" and the reference to the node has been put to the hosting machine.

Once the Pods are created, the dependencies satisfied and the volume claims bound, then the deployment finishes successfully and all Pods are in a running and ready state. We have now a cluster made of one physical node hosting all the elements of a cluster, with a 5G Core running on top of it made of different components interconnected one with another and waiting for requests from user devices:

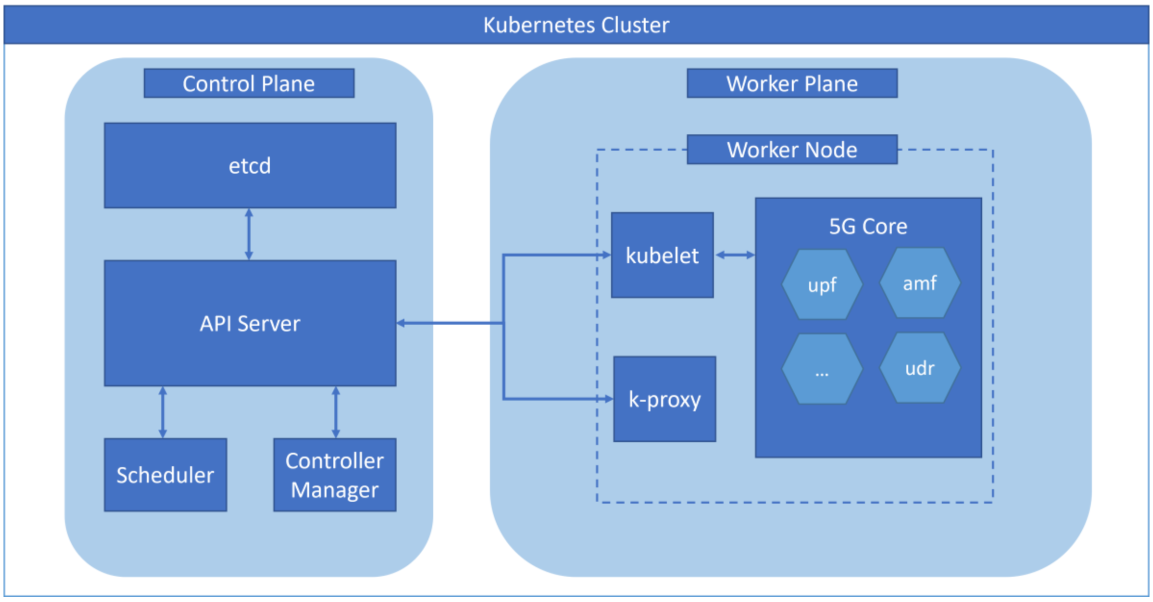
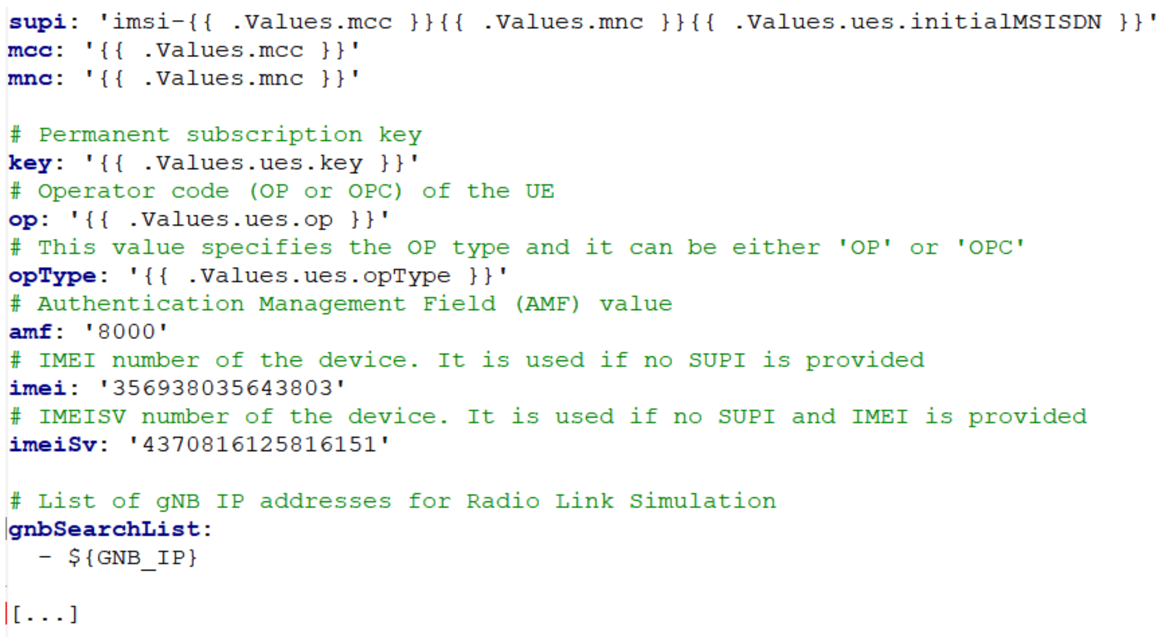


Figure 38 - Solution Architecture

4.4 UERANSIM Deployment

UERANSIM is an open-source implementation of 5G UE and RAN (gNodeB) and to put it simply, it can be considered as a combination of a 5G mobile phone and a base station. The "SIM" at the end of the term stands for simulated, this implementation is used to simulate mobile phone devices and Radio



```
supi: 'imsi-{{ .Values.mcc }}{{ .Values.mnc }}{{ .Values.ues.initialMSISDN }}'
mcc: '{{ .Values.mcc }}'
mnc: '{{ .Values.mnc }}'

# Permanent subscription key
key: '{{ .Values.ues.key }}'
# Operator code (OP or OPC) of the UE
op: '{{ .Values.ues.op }}'
# This value specifies the OP type and it can be either 'OP' or 'OPC'
opType: '{{ .Values.ues.opType }}'
# Authentication Management Field (AMF) value
amf: '8000'
# IMEI number of the device. It is used if no SUPI is provided
imei: '356938035643803'
# IMEISV number of the device. It is used if no SUPI and IMEI is provided
imeiSv: '4370816125816151'

# List of gNB IP addresses for Radio Link Simulation
gnbSearchList:
  - ${GNB_IP}

[...]
```

Figure 39 - UE Configuration File

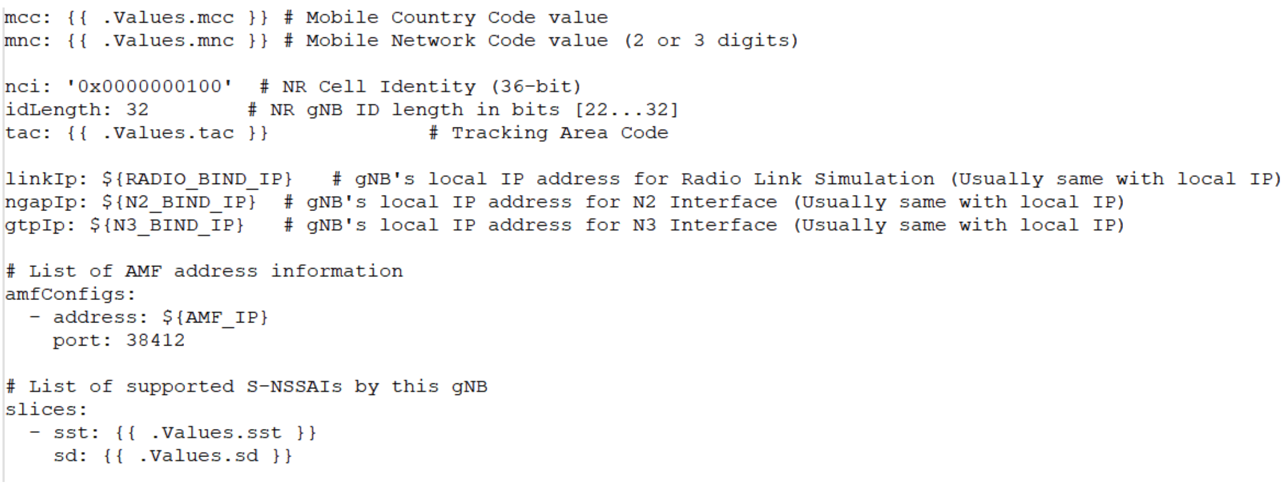
Access Network to test the 5G Core Network [21]. It is possible to have several deployment options when it comes to the simulated UE and RAN. It is possible to have UEs on bare metal while having a software defined version of the RAN or have both elements implemented as software to be hosted in a container or a virtual machine. In this project, the UERANSIM has been deployed using helm charts and hosted on the Kubernetes cluster and all the components are implemented in software. The scenario includes a gNodeB and 4 UEs that send requests to the cluster. Both the UE and gNodeB have YAML configuration files to define the parameters to be used in the deployment. The following image shows the most interesting parts of these files:

An important thing to notice is the first field called *supi*, a supi defined a UE and is made of different values put together:

IMSI = [mcc | mnc | msidsn] (In total 15 digits)

Where mcc and mnc are the exact same fields mentioned in the AMF configuration, in order to have a successful communication between the UERANSIM and 5G Core, these values must be the same in both deployments. The msidsn value can be set to "0000000001". Another important field is the *gnbSearchList*, this must be set to the IP address of the gNodeB otherwise the UE will not be able to communicate with it.

When it comes to the gNodeB, the configuration file will be as follows:

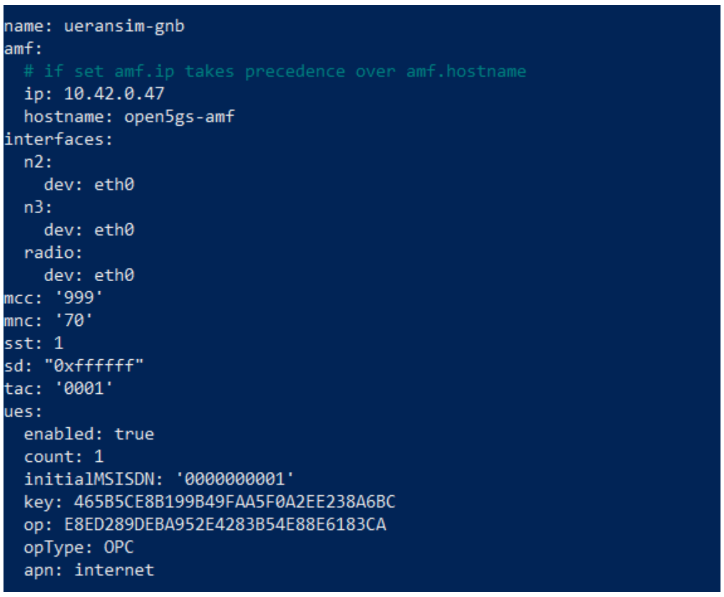


```
mcc: {{ .Values.mcc }} # Mobile Country Code value
mnc: {{ .Values.mnc }} # Mobile Network Code value (2 or 3 digits)

nci: '0x0000000100'  # NR Cell Identity (36-bit)
idLength: 32        # NR gNB ID length in bits [22...32]
tac: {{ .Values.tac }}              # Tracking Area Code

linkIp: ${RADIO_BIND_IP}   # gNB's local IP address for Radio Link Simulation (Usually same with local IP)
ngapIp: ${N2_BIND_IP}  # gNB's local IP address for N2 Interface (Usually same with local IP)
gtpIp: ${N3_BIND_IP}   # gNB's local IP address for N3 Interface (Usually same with local IP)

# List of AMF address information
amfConfigs:
  - address: ${AMF_IP}
    port: 38412

# List of supported S-NSSAIs by this gNB
slices:
  - sst: {{ .Values.sst }}
    sd: {{ .Values.sd }}
```

Figure 40 - GNB Configuration File

Both mcc and mnc fields are set here as well as in the UEs and AMF. The IP address of gNodeB will be automatically collected when the deployment is performed and set to the IP address of the Pod hosting it. A very important configuration section is amfConfigs, here are set both the IP address of the AMF and the port on which it is listening. Without these parameters it would not be possible to communicate with the Core. Please note that AMF is the only component of the 5G Core that is reached by the UERANSIM. The UE will connect to the gNodeB and it will in turn send the requests to the core.

As in the 5G Core deployment, here we have a general configuration file too. It provides values to the placeholders in the ue.yaml and gnb.yaml files shown above:



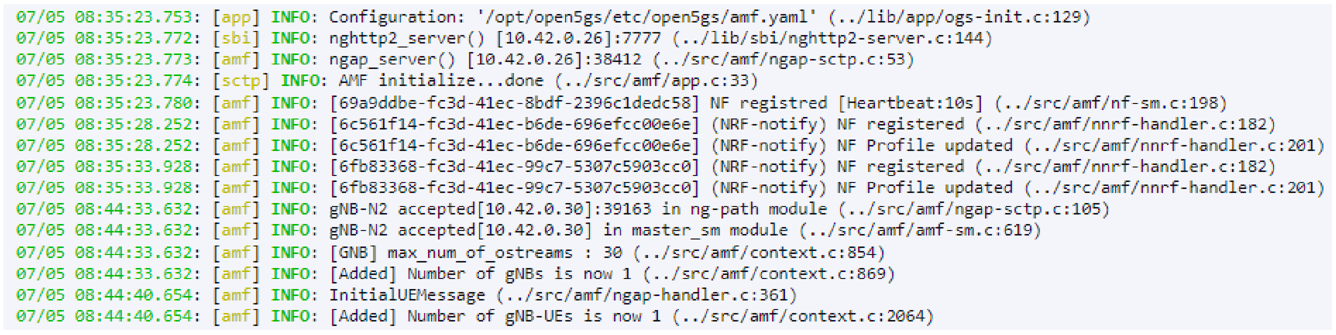
```
name: ueransim-gnb
amf:
  # if set amf.ip takes precedence over amf.hostname
  ip: 10.42.0.47
  hostname: open5gs-amf
interfaces:
  n2:
    dev: eth0
  n3:
    dev: eth0
  radio:
    dev: eth0
mcc: '999'
mnc: '70'
sst: 1
sd: "0xffffff"
tac: '0001'
ues:
  enabled: true
  count: 1
  initialMSISDN: '0000000001'
  key: 465B5CE8B199B49FAA5F0A2EE238A6BC
  op: E8ED289DEBA952E4283B54E88E6183CA
  opType: OPC
  apn: internet
```

Figure 41 - UE and gNodeB general configuration file

The deployment can be started with the same command used for the Core deployment:

helm install <deployment_name> <charts>

The result will be two Pods, one hosting the gNodeB and another one the UEs, all the user equipment created will be hosted within the same Pod and the devices will send requests to the Core. First of all, gNodeB registers to AMF and then UE send requests to the core, they get authenticated and register to it. The following image shows the logs of AMF showing the registration of gNodeB and requests from UE:



```
07/05 08:35:23.753: [app] INFO: Configuration: '/opt/open5gs/etc/open5gs/amf.yaml' (../lib/app/ogs-init.c:129)
07/05 08:35:23.772: [sbi] INFO: nghttp2_server() [10.42.0.26]:7777 (../lib/sbi/nghttp2-server.c:144)
07/05 08:35:23.773: [amf] INFO: ngap_server() [10.42.0.26]:38412 (../src/amf/ngap-sctp.c:53)
07/05 08:35:23.774: [sctp] INFO: AMF initialize...done (../src/amf/app.c:33)
07/05 08:35:23.780: [amf] INFO: [69a9ddbe-fc3d-41ec-8bdf-2396c1dedc58] NF registred [Heartbeat:10s] (../src/amf/nf-sm.c:198)
07/05 08:35:28.252: [amf] INFO: [6c561f14-fc3d-41ec-b6de-696efcc00e6e] (NRF-notify) NF registered (../src/amf/nnrf-handler.c:182)
07/05 08:35:28.252: [amf] INFO: [6c561f14-fc3d-41ec-b6de-696efcc00e6e] (NRF-notify) NF Profile updated (../src/amf/nnrf-handler.c:201)
07/05 08:35:33.928: [amf] INFO: [6fb83368-fc3d-41ec-99c7-5307c5903cc0] (NRF-notify) NF registered (../src/amf/nnrf-handler.c:182)
07/05 08:35:33.928: [amf] INFO: [6fb83368-fc3d-41ec-99c7-5307c5903cc0] (NRF-notify) NF Profile updated (../src/amf/nnrf-handler.c:201)
07/05 08:44:33.632: [amf] INFO: gNB-N2 accepted[10.42.0.30]:39163 in ng-path module (../src/amf/ngap-sctp.c:105)
07/05 08:44:33.632: [amf] INFO: gNB-N2 accepted[10.42.0.30] in master_sm module (../src/amf/amf-sm.c:619)
07/05 08:44:33.632: [amf] INFO: [GNB] max_num_of_ostreams : 30 (../src/amf/context.c:854)
07/05 08:44:33.632: [amf] INFO: [Added] Number of gNBs is now 1 (../src/amf/context.c:869)
07/05 08:44:40.654: [amf] INFO: InitialUEMessage (../src/amf/ngap-handler.c:361)
07/05 08:44:40.654: [amf] INFO: [Added] Number of gNB-UEs is now 1 (../src/amf/context.c:2064)
```

Figure 42 - Successfult Deployment and Connection Logs

UERANSIM provides a TUN interface to enable UEs to have internet connectivity, and this is done for each PDU session. Once a session is successfully established, the UE creates a TUN interface and a routing table along with an IP route. Then it is possible to test this with a ping: ping -I uesimtun1 google.com

Once the deployment has been successfully completed, we will have the following solution structure:

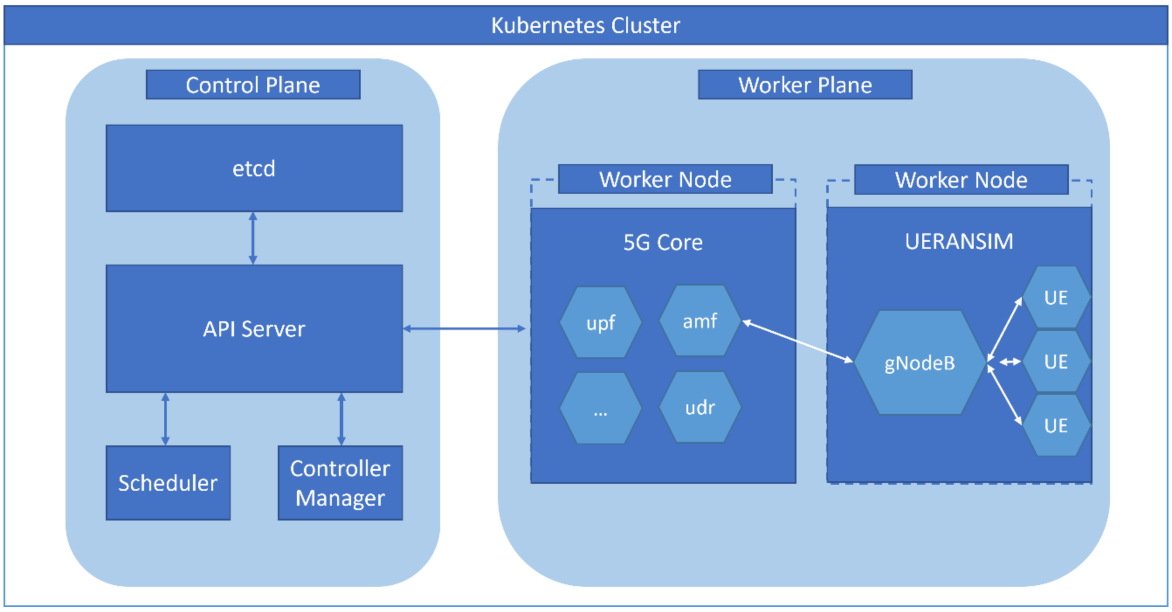


Figure 43 - Solution with UERANSIM

This chapter explained how a deployment of the 5G system can be performed on a Kubernetes cluster. It is now possible to perform different tests on the infrastructure and collect data about the cluster and its behaviour when subjected to different scenarios. The next chapter shows the test cases and the collected results.

5. Test Cases and Results

This chapter is dedicated to lay out some tests performed on the cluster and their outcome. First, it describes the behaviour and recovery time of the cluster in case of failures and speed of scaling. Then, it compares two different deployments scenarios of the 5G Core– one in full version and one in a more essential version to represent an industry 4.0 scenario. A third test has been performed to gather deployment measurements in a smart city scenario with another different structure of the 5G Core.

5.1 Case 1 – Failure and Scaling

This test case aims to understand the reaction of the cluster to a (or a set of) failure(s) of Pods and to a scenario of scaling up. Because of several reasons, it may happen that some pods face a failure in their lifecycle. But as we know, Kubernetes is robust to failures. However, we need to understand how it responds to these types of events and have a measurement of the time it would need to first understand that there was a problem and then recover from it. Thanks to its continuous monitoring of the current state of the cluster, it was noticeable that K8s promptly reacts to a failure. When it comes to the time needed to the cluster to recover from such failures, the below chart shows the measurements for each element of the core:

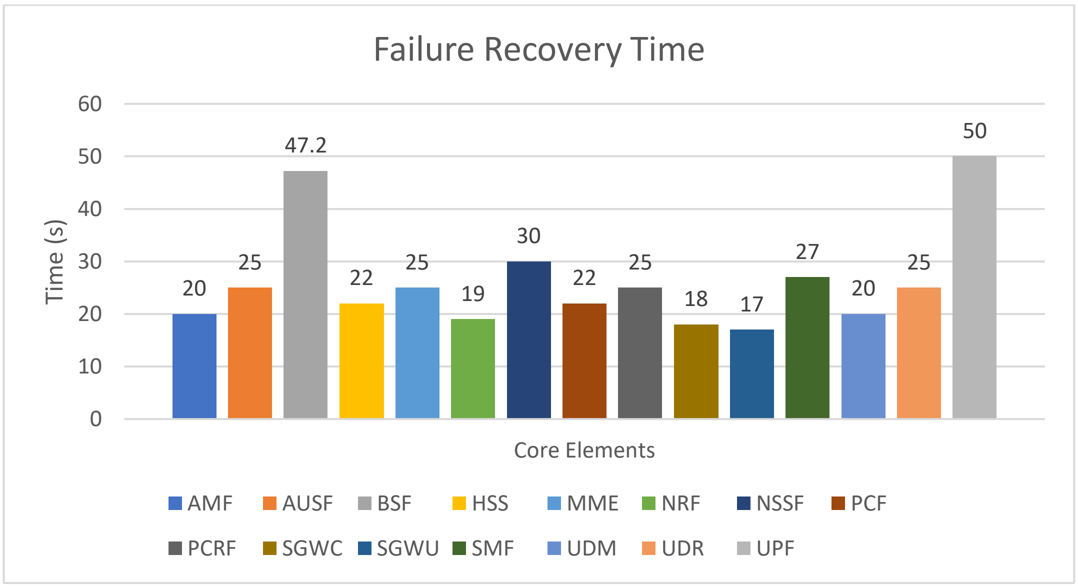


Figure 44 - Failure Recovery Times of the Cluster

When a Pod fails, Kubernetes promptly notices its failure and starts a back-up Pod that will replace the failed one on the node. The time needed to create a new Pod for each element is shown in the charts and as we can see it requires, for most of the elements, less than 30 seconds to have a fully running back-up. The only elements requiring more time compared to the rest of the core are the BSF and USF. Whilst a Pod is running and there are some issues, the kubelet is able to restart containers to handle the faults. Within a Pod, Kubernetes tracks different container states and determines what action to take to make the Pod healthy again.

In order to respond to high volumes of requests or to have flexibility, it may be required to scale up some elements of the core. This may also be applied only for certain periods of time when the quantity of required resources is high and once that period of time is over, it is possible to scale down again. Considering having more replicas of the elements that may require scaling, the below chart shows the time needed to perform the action. The scaling times are for obvious reasons lower than the failure recovery times. The only element that requires a time much higher than the other elements is again the BSF, but this is understandable as it has dependencies to manage, i.e. with the nrf.

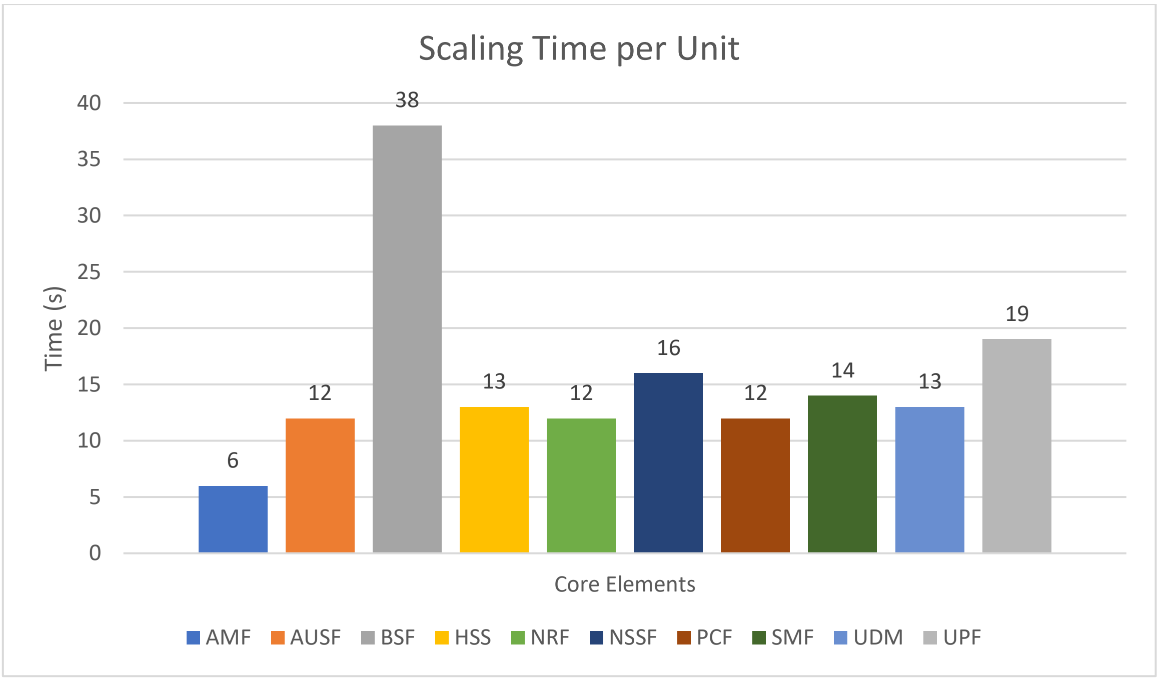


Figure 45 - Scaling Time per Unit of the Cluster

5.2 Case 2 – Deployments Comparison

This test case compares two different configurations of the core, one in the full format made of all the components that have been used in the whole project and explained already in the previous chapters, and another one made of only the main elements needed for the core to work in a more essential scenario. An example of where the second core could be deployed is an *Industry 4.0* scenario, here the infrastructure is static, does not require mobility and the communication can be achieved with the essential core. In an industrial set-up, usually the equipment is always the same, the mobility is very limited so a core with only the main functional blocks works very well. The solution contains elements such as: AMF. AUSF, NRF, NSSF, PCF, UDM, UDR, SMF and UPF.

The below image shows a comparison between the two deployments in terms of time needed for the deployment, number of Pods created in the cluster, the cores and memory consumption:

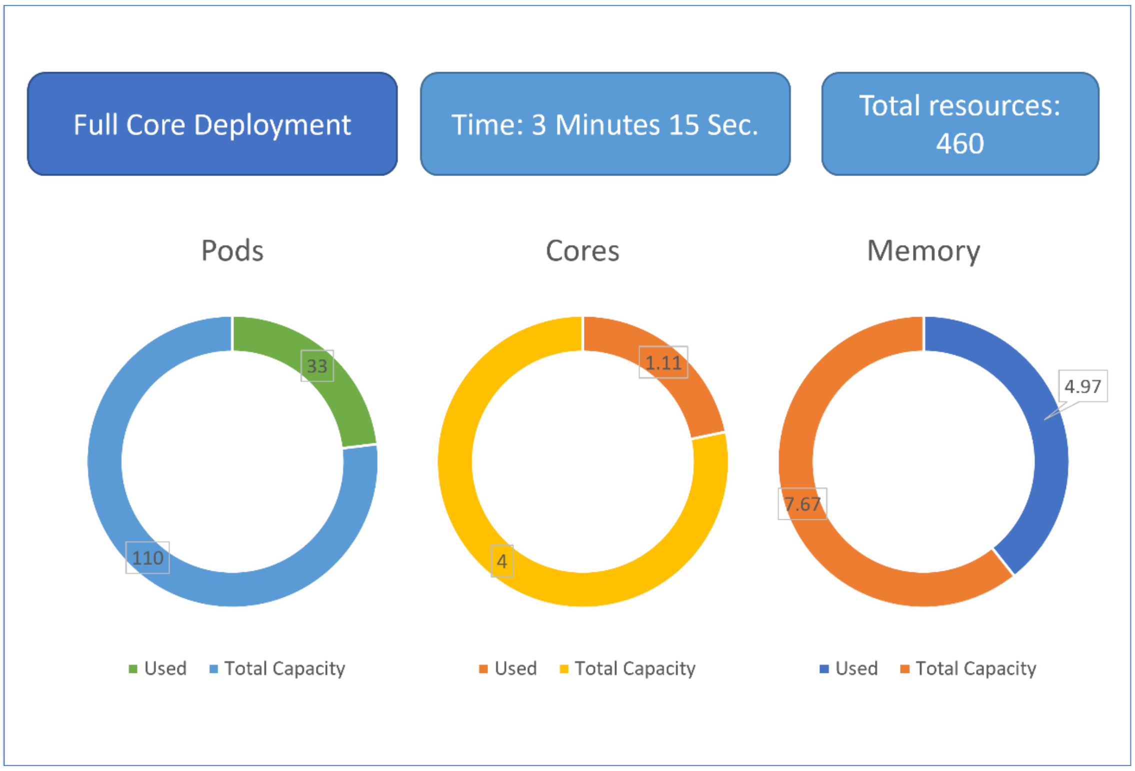


Figure 46 - Full Core Deployment Measurements

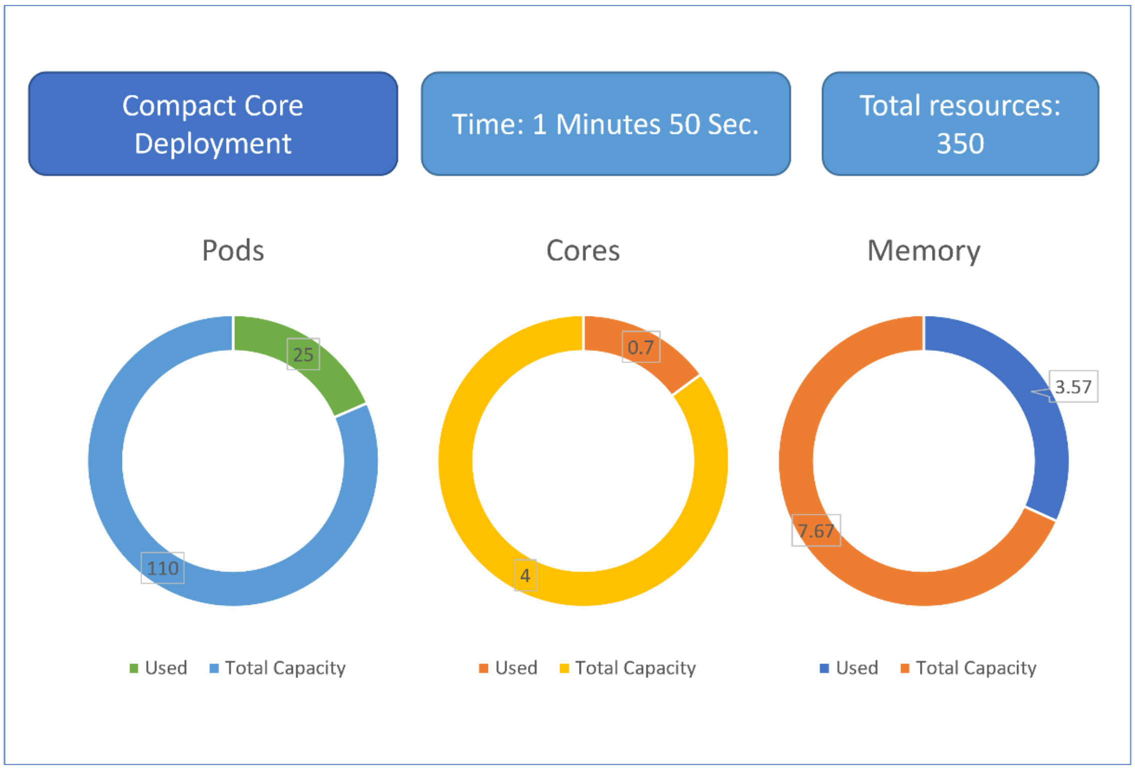


Figure 47 - Compact Core Deployment Measurements

The time needed to deploy the Compact Core is nearly half compared to the one needed for a full version of the core to be deployed. When it comes to the resources instead, the number of Pods reduces in the second case but not as much as one would expect. In reality, this number does make sense as there are several Pods working in the background to support the deployment. So, the number decreases only by the elements not needed in the compact deployment. The amount of processing capacity that is needed in the second deployment is 0.4 units less compared to the first deployment. The same applies to the memory, it falls from 4.97 GBs to 3.57GBs. This second deployment would certainly be useful in case the resources at disposal are limited and the functional requirements are the standard ones.

5.3 Case 3 – Smart City

Smart Cities are characterised by huge numbers of devices in diverse contexts such as IoT, eMobility, Services, Smart building etc. requiring great mobility, flexibility and scalability from the network infrastructure. In these types of scenarios there is a huge quantity of devices connecting to the network and on a specific moment there can be hundreds of thousands of devices connected and communicating data at the same time. These high numbers of devices become even bigger if we consider IoT and sensors distributed all over the city. The devices require high mobility while the sensors need great throughput and the capacity to manage huge quantity of data. To enable a support for these diverse contexts, 5G introduces the concept of network slicing. By slicing a physical network into several logical networks, each one can provide tailored services for a distinct application scenario. 5G network slices represented by logically isolated and self-contained networks are flexible enough and highly customizable to accommodate diverse business-driven use cases simultaneously over the same network infrastructure [22].

To support these requirements, the network must adapt to a less conventional structure and provide replication to resources that can manage big quantities of requests, support mobility and manage big quantities of data. This test case re-creates a scenario like the one mentioned above and provides a deployment of the 5G core that differs in terms of its control plane and user plane configuration. Some elements of the core have been scaled up to enable the flexibility and mobility of the infrastructure. In particular, replication has been applied to AMF and SMF in the control plane while in the user plane to the UPF. The huge quantity of user equipment connecting to the core requires replication of these elements, especially UPF. The below image shows the structure of the deployment:

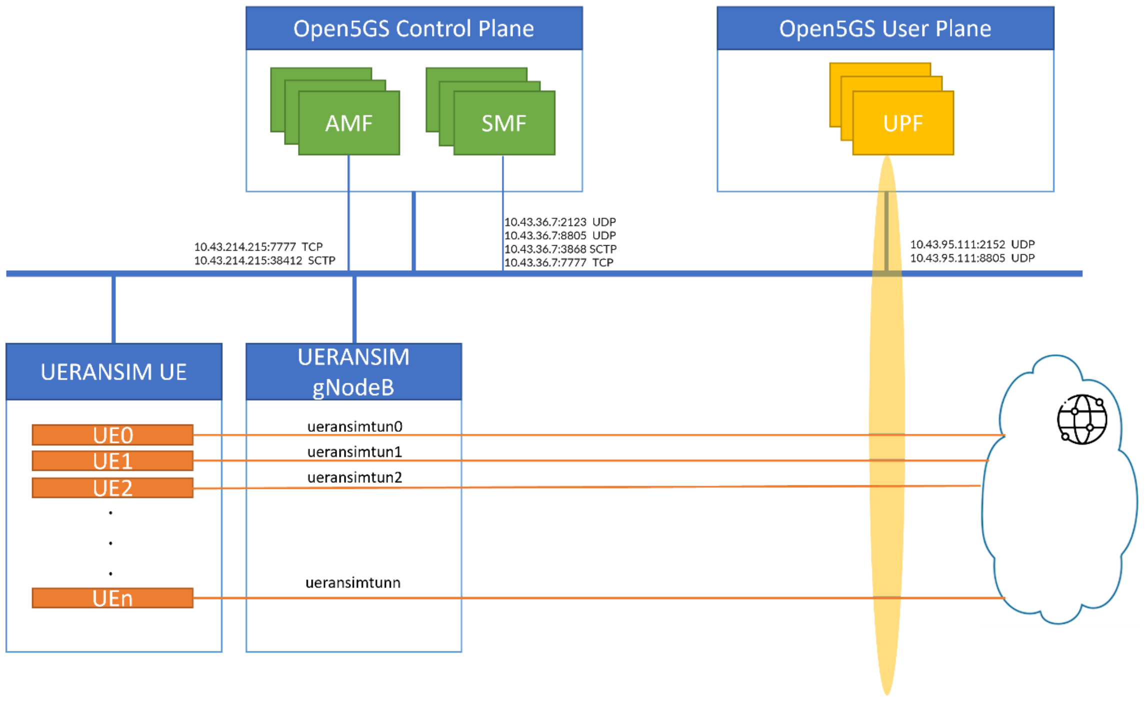


Figure 48 - Core Deployment in a Smart City Scenario

In this test case, there is a replication factor of 3, 5 and 10 for AMF, SMF AND UPF. The below images show in each case the time needed to the deployment, the resources consumed in terms of created Pods, cores and memory consumption:

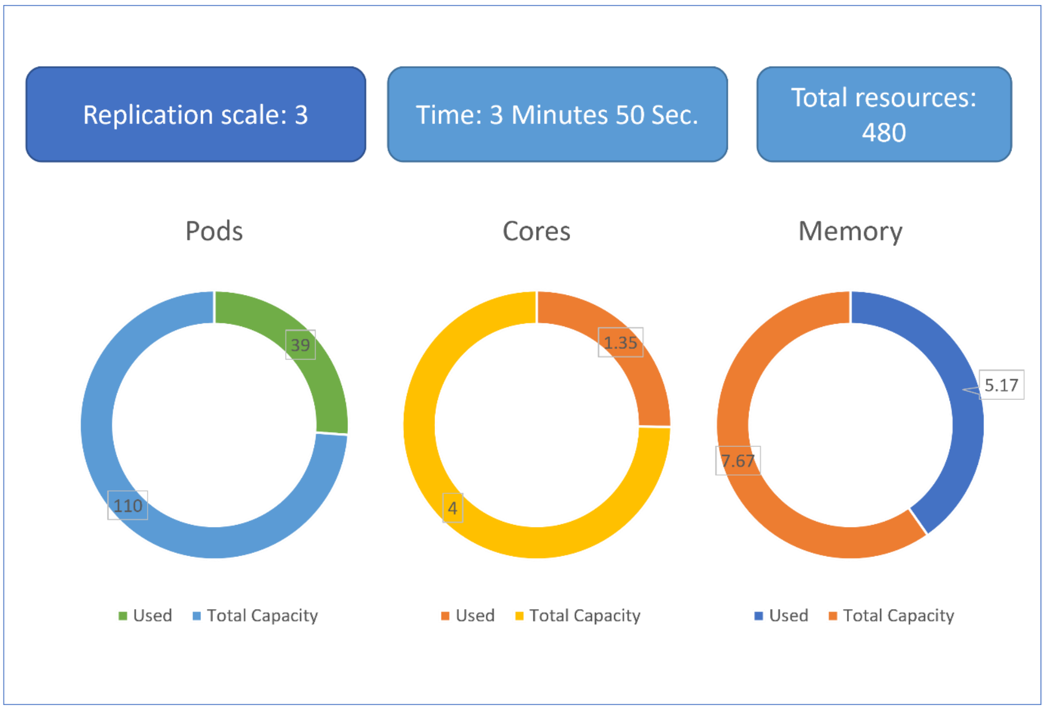


Figure 49 - Replication Degree 3

The time needed for the deployment of with the case with replication degree 3 is a little higher compared to the full core without replication. The number of pods goes up to 39 compared to the 33 as 6 more replicas were added to get to 3 replicas for each element. The core consumption goes up to 1.35 instead of the previous 1.11. The same applied to the memory that reaches 5.17 which is the 67.4% of total available memory.

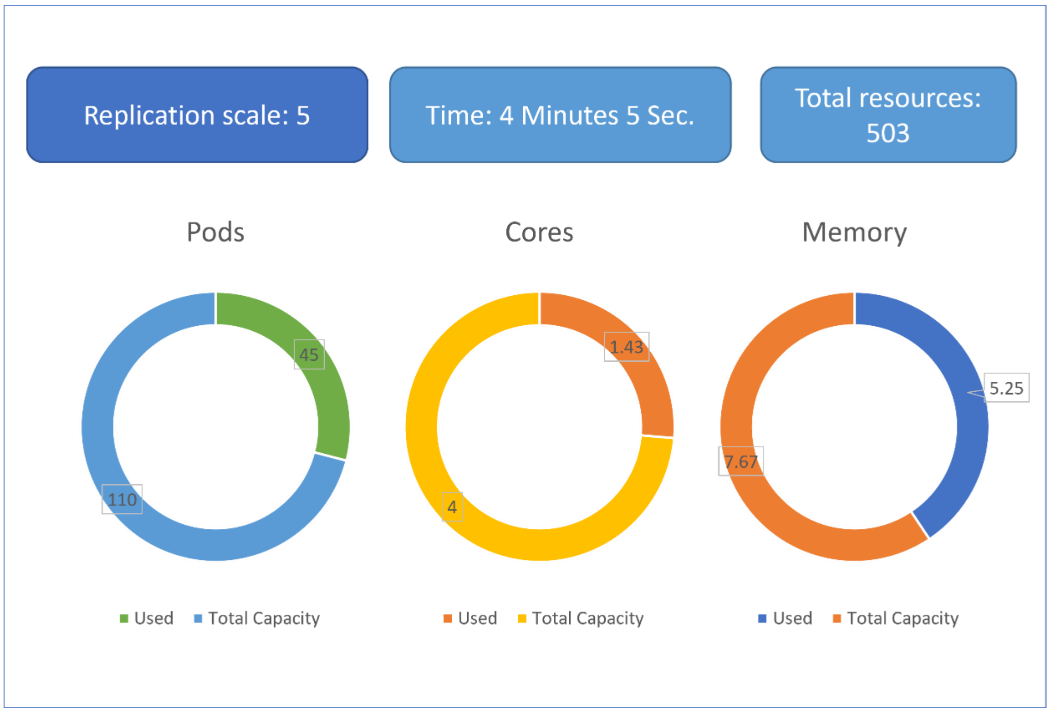


Figure 50 - Replication Degree 5

The resource consumption trend goes up in all cases as the test case required the deployment of a higher number of components. As shown in the image above, the number of Pod has gone up to 60 while the core consumption reached 1.55 but still there is great margin as there are 4 cored dedicated to the cluster. The memory has gone up to 5.25 but the difference compared to the previous value is only 0.08GB which is not a very high number.

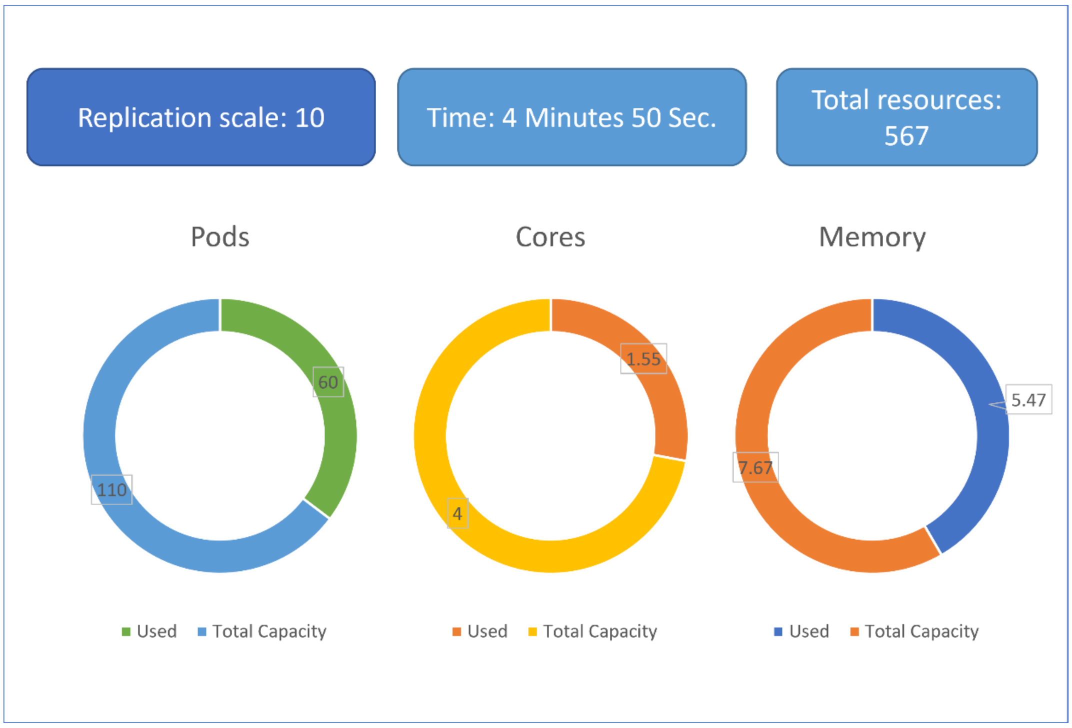


Figure 51 - Replication Degree 10

The last measurement instead is for a replication degree of 10, so all three elements have 10 replicas working in the cluster in the control and user plane. The time needed to finish the deployment has gone up to 4 minutes and 50 seconds, a considerable change compared to the difference of deployment time in the first two cases, but still overall it is an acceptable time for the deployment of a network core. The processing resources usage has gone up to 1.55 but it is very much acceptable as a value. The memory instead has reached a value of 5.47 which is a little high, in total the memory usage is 71.3%. So, in case there were other replicas to be instantiated, or more applications to be imported in the cluster, memory could become an issue to resolve by allocating extra memory to the cluster.

These test cases have allowed to observe the behaviour of both the cluster and the core under different circumstances and understand, from the point of view of a provider, how things evolve in different scenarios. In particular, the first test case highlights the recovery of the system in case of failures and in case scaling is required. In the second test case it is interesting to see how an industrial core deployment differs from a full core deployment in terms of time and resource consumption. In the third and last test case instead, the structure has changed and become more complex is respect to a normal full core deployment. Here, the replication is applied to only some modules of the core allowing it to be more flexible. In this case, like in the second test case, it was possible to observe the time needed to perform the deployment along with the resource consumption but in 3 different replication degrees.

6. Conclusions and Future Work

This research and project work aimed to find an alternative the to existing 5G Core deployment solutions, moving towards a true cloud native approach to be applied rather than a pseudo-cloud native one based on the distribution and hosting of MANO implementations on the cloud. The idea was to use Kubernetes as an implementation of MANO standard opposed to other pre-existing solutions. As a result, we can conclude that Kubernetes cannot just act as a NFV Infrastructure and VIM but also as an orchestrator and manager of Virtualised Network Functions. The achievement of this project was to have a deployment of 5G Core on top of a Kubernetes cluster and managing the deployment, lifecycle, failure recovery, scaling, workload balancing etc. of VNFs directly by Kubernetes. The outcome has allowed to understand how this technology is complete and suitable for the above-mentioned purposes.

We started by understanding which technologies and paradigms allowed to move towards the concept of software defined networking and which standards have been put in place for this concept. It was then important to understand very well the structure of 5G networks and how they differ from the previous ones. In order to create deployments and test, it was necessary to comprehend what scenarios required the introduction of a new generation of telecommunications. Then, an in-depth study of the existing implementations and solutions allowed to define the points where improvement could be made. Gathering knowledge about novel but still well implemented and supported technologies allowed to create a roadmap to introduce the cloud native concept into the 5G Networks.

We now have the 5G Core deployed on a Kubernetes cluster and managed entirely by it. The core has been tested with UE and RAN defined in software. We have a set of different configurations of the core that can be deployed on a Kubernetes cluster depending on different scenarios that we want to test. This allowed to understand how the cluster reacts in the different roles mentioned above. Especially how it recovers from failures and how it reacts to the scaling of hosted components. Applying the concept of differentiated services for different usages, allowed to create tests for core deployment in an industrial context and a smart city one. The results showed how dynamic the configuration of the core can become to cover these diverse set-ups and how efficiently Kubernetes can manage these changes with very low variations in terms of time of deployment and resource consumption.

In the future, this work can be certainly extended under various aspects. As for now, the infrastructure has been tested from the prospective of a provider. One possible work could be focused on enriching the user side, UE and RAN, and study the deployment under different scenarios. This would allow to understand how the cloud native 5G Core reacts to different requirements, i.e. huge quantity of requests from as many devices in a smart city scenario. A second extension could consist in the comparison of Open5GS with other implementations of the 5G Core such as Free5GC and Magma. In this way, it would be possible to measure the time that is needed for the core to be deployed, the performances against failures and scaling requirements. Moreover, this would allow to gain knowledge about the limitations of these implementations and understand which are the applications where each of them is better compared to others. This comparison would enable dedicated core deployments for different scenarios to fulfil at best the requirements.

Another extension to this work could be to have a multi-cluster solution as Kubernetes would be able to manage a multi-cluster scenario. In this way the distribution of the core could enable huge flexibility, dynamicity and scaling capabilities.

These are some ideas of possible future works that can be performed taking into consideration this thesis project work. Several other options can be found to extend these concepts now that a first solution is available.

References

[1] Wang, An; Guo, Yang; Hao, Fang; Lakshman, T.; Chen, Songqing (2 December 2014). "Scotch: Elastically Scaling up SDN Control-Plane using vSwitch based Overlay". ACM CoNEXT

[2] Trends on virtualisation with software defined networking and network function virtualisation - Lorena Isabel Barona López,Ángel Leonardo Valdivieso Caraguay,Luis Javier García Villalba,Diego López.

[3] Network Functions Virtualisation (NFV); Architectural Framework – ETSI Group Specification.

[4] Network Functions Virtualisation (NFV); Management and Orchestration – ETSI Group Specification.

[5] Kubernetes [Online] - https://kubernetes.io

[6] S. Parkvall, E. Dahlman, A. Furuskar, and M. Frenne, "NR: The New 5G Radio Access Technology," IEEE Communications Standards Magazine, vol. 1, no. 4, pp. 24–30, December 2017.

[7] Open, Programmable, and Virtualized 5G Networks: State-of-the-Art and the Road Ahead. Leonardo Bonati, Michele Polese, Salvatore D'Oro, Stefano Basagni, Tommaso Melodia Institute for the Wireless Internet of Things, Northeastern University, Boston, MA 02115, USA

[8] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," Internet Things J., vol. 3, pp. 637–646, 2016.

[9] Internet of Things statistics for 2022 - Taking Things Apart. [Online] https://dataprot.net/statistics/iot-statistics/

[10] Service-Based Architecture for 5G Core Networks – Gabriel Brown, Heavy Reading.

[11] What is 5G Network Architecture – Herald Rammet, Digi International – 2021

[12] Analysis for Comparison of Framework for 5G Core Implementation. Francisco Joaquim de Souza Neto, Edson Amatucci, Nadia Adel Nassif, Pedro Augusto Marques Farias – ICISCT 2021.

[13] Open5GS [Online] https://open5gs.org/open5gs/docs/guide/01-quickstart/

[14] Free5GC, “Roadmap of free5GC project,” Free5GC, 13 January 2019. [Online]. Available: https://www.free5gc.org/roadmap. [Acesso em 10 June 2021].

[15] Magma [Online] https://docs.magmacore.org/docs/basics/introduction

[16] Toward True Cloud Native NFV MANO. David Breitgand, Vadim Eisenberg , Nir Naaman, Nir Rozenbaum , Avi Weit - 2021 IEEE

[17] Rancher - https://rancher.com/why-rancher

[18] Calico by Tigera – [Online] https://projectcalico.docs.tigera.io/about/about-calico

[19] Helm – [Online] https://helm.sh/docs/intro/quickstart/

[20] UERANSIM Configurations - [Online] github.com/aligungr/UERANSIM/wiki/Configuration

[21] UERANSIM Implementation – [Online] https://github.com/aligungr/UERANSIM

[22] An Overview of Network Slicing for 5G - Shunliang Zhang. IEEE Wireless Communications • June 2019