Containerization and Services Lifecycle Management

Are We There Yet?

A technical paper prepared for presentation at SCTE TechExpo24

Nasir Ansari
Network Architect
Rogers Communications Inc.
Nasir.Ansari@rci.rogers.com

Yassar Abbas, Rogers Communications Inc.

Table of Contents

Title **Page Number**

1. Introduction 4
2. Current Appliance Based CCAP 4
    2.1. Hardware 4
    2.2. Software 4
3. Distributed Access Architecture 5
    3.1. Cloud-Native vCCAP 6
    3.2. Converged Interconnect Network 6
    3.3. Remote-PHY Device (RPD) 8
4. Kubernetes (K8s) Cluster Architecture 8
    4.1. Kubernetes Services Controller 10
    4.2. Kubernetes Worker Nodes 12
    4.3. Cloud Control Manager (Informational) 13
5. Container as a Service (CaaS) 13
6. CaaS & Application Combined 14
7. (vCCAP) Deployment Model 1 14
    7.1. Linecard functions in a Container 15
    7.2. Kubernetes Cluster View 15
8. (vCCAP) Deployment Model 2 15
    8.1. Mixing it up (Application/Kubernetes System) 16
9. (Wireline) Considerations/Learnings 16
    9.1. Separation of Platform and Application Configuration 16
    9.2. Automation 17
    9.3. Server, Platform and Application/Service Management 17
    9.4. Command Line Interface -> Programmatic Interfaces 17
    9.5. Platform/Application related considerations 17
10. (Potential) Future Changes to (Wireline) Deployment Model 17
11. Wireless Network Functions Virtualization 17
    11.1. Virtualization Background 17
    11.2. Wireless Core deployment Models 19
        11.2.1. Centralized Deployment Model 19
        11.2.2. Edge Deployment 20
    11.3. Life Cycle Managemnt 23
12. Conclusion 24

Abbreviations 26

Bibliography & References 27

List of Figures

Title **Page Number**

Figure 1 - Generic Hardware / Software Depiction of Appliance 5
Figure 2 - Distributed Access Architecture 6
Figure 3 - Spine-Leaf Switches 7
Figure 4 - Metro-Aggregation Router 7
Figure 5 - Kubernetes Cluster Architecture 8
Figure 6 - CaaS Architecture 13
Figure 7 - K8s/vCCAP Deployment Model 1 15
Figure 8 - K8s/vCCAP Deployment Model 2 16

Figure 9 - Wireless Core Transition path. 18
Figure 10 - Wireless Core Centralized K8s Deployment Model 20
Figure 11 - Distributed K8s cluster Deployment Model 21
Figure 12 - K8s Cluster Edge Deployment Model 22
Figure 13 - K8s Cluster compact Deployment Model 23

1. Introduction

Today's CCAP/CMTS Platforms' have been continuously improved over time. Platform redundancy has also been added to increase resiliency. However, all connected Subscribers "home in" to a single Physical location. This paper discusses Containerization and Services Lifecycle Management Models, Architecture and lessons learnt from early field trials of Virtual CCAP Technology and Production Deployments of Wireless Core Services to accomplish distribution of (Network) Functions to multiple Physical locations.

Next Generation Technology Platforms (for both Wireline and Wireless Cores) are evolving. Today's Monolithic Platforms are being replaced by Distributed Services/Functions running on Docker/Containers. This Journey can be broken up into phases based on Platform and/or Operator readiness.

Currently most implementations use Containers running on "Bare Metal". Implementation Options include Separation of Services' Controller and Service Instantiation or running everything on the same "Cluster". Services/Functions could be amalgamated or broken out into multiple Micro-Services. There could be an instance for each function or multiple instances. There are Configuration, Operational, Load-Balancing and Failover considerations for each of the Models.

This paper will show how Distribution of Access Network Gateway Functions can further improve Service Delivery, Resiliency, Management and Operations.

2. Current Appliance Based CCAP

Today's Appliance Based CCAPs have developed and matured over the last 10-15 years. The "All-in-One" Boxes typically have a Network Side Interface (NSI), Downstream Side Interface (DSI) facing the Customer, Controller/Supervisor Functionality coupled with Packet and Radio Frequency (RF) Switching to direct Traffic to Different Modules. Each aspect of the Appliance has a defined grouping of Functions.

2.1. Hardware

The Modules within the Appliance will either have Application Specific Integrated Circuits (ASICs) for Functions requiring Throughput/Performance or General-Purpose Units (GPUs) for Controller related Functionality.

In some cases, modules may be based on Field Programmable Gate Arrays (FPGAs) that provide a combination of Controller Logic and Throughput.

2.2. Software

From a functional perspective the Appliance can generically be represented by the following Figure:

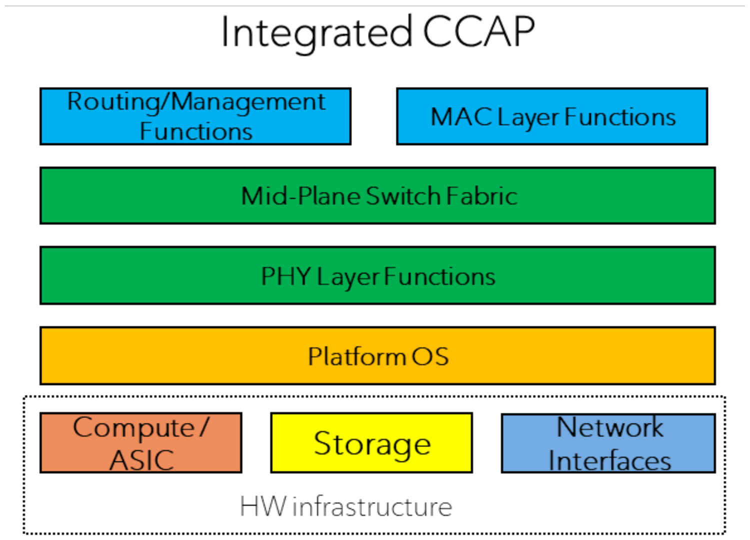


Figure 1 - Generic Hardware / Software Depiction of Appliance

Each Module within the Appliance may follow the same pattern. There would typically be larger blocks of code and hence the "all-in-one" description.

3. Distributed Access Architecture

The Integrated Appliance is broken up into Hardware (Physical Layer) Components and Software (Logical Layer) Components/Services. The "Mid-Plane" switching is replaced by the Converged Interconnect Network (CIN). The following Figure shows this Distribution of functionality:

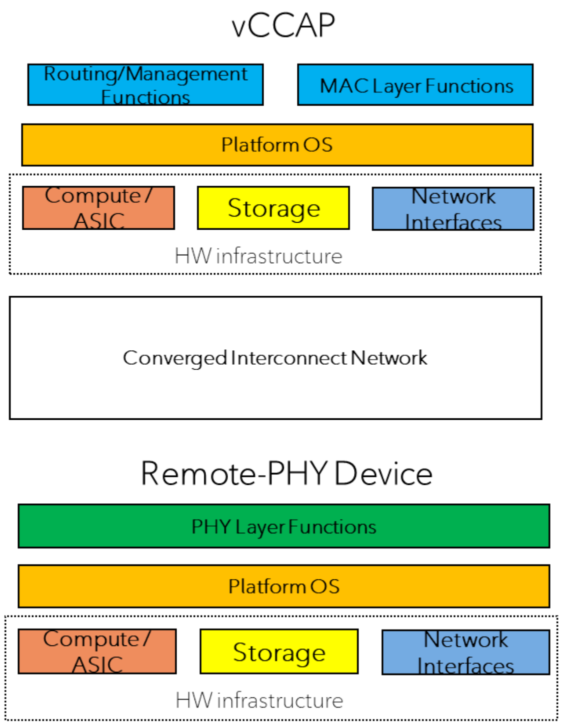


Figure 2 - Distributed Access Architecture

3.1. Cloud-Native vCCAP

The Software Components are broken up into smaller blocks of code. This break up into "micro-services" makes for more portable components that can each be treated separately for improvements or bug fixes. Each micro-service runs within its own Container. Each container has resource assignments.

3.2. Converged Interconnect Network

The Converged Interconnect Network (CIN) is based on a "Spine-Leaf" (SL) Switch Architecture and provides Many-Many connectivity between vCCAPs and Remote-PHY Devices (RPD). A variation on the SL Switches would be Metro and Aggregation Router/Switches. The Metro-Aggregation Router/Switches are "heavier" on Routing functionality. The SL Switches tend to be "lighter" on Routing functionality.

The following Figure shows the generic Spine-Leaf Switch deployment.

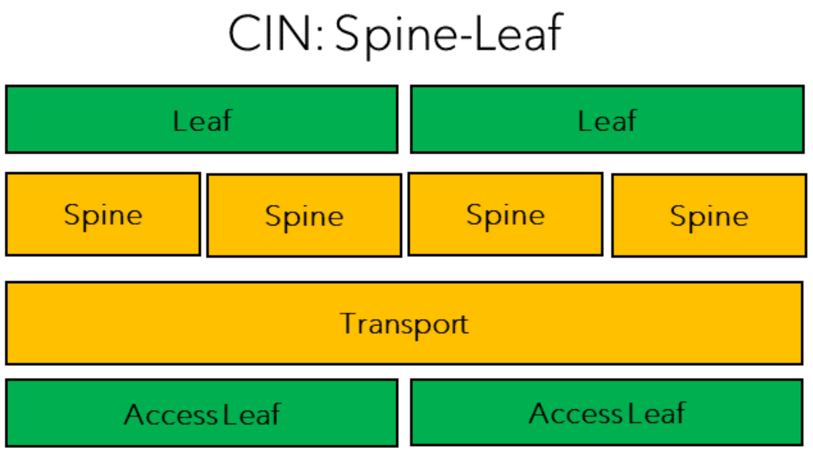


Figure 3 - Spine-Leaf Switches

The following Figure shows the generic Metro and Aggregation Router/Switch deployment.

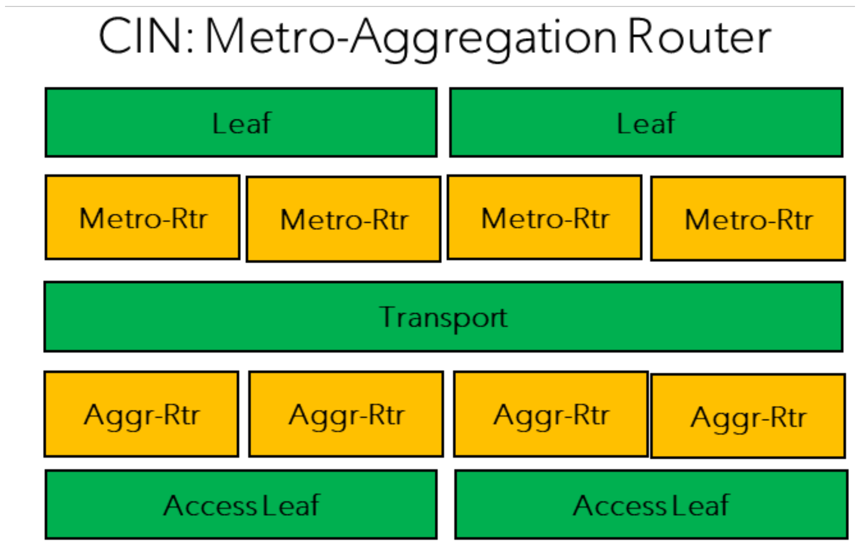


Figure 4 - Metro-Aggregation Router

The Architecture Options available for the CIN are:

1. EVPN/VxLAN
2. MPLS or SRv6

With the introduction of a suitable Optical Transport Network there is flexibility to create:

1. Point-to-Point -or-
2. Multipoint-to-Multipoint Topologies.

Later, in this paper the Authors will show how this added functionality can be an enabler for improvements in Services' Availability.

3.3. Remote-PHY Device (RPD)

All Physical (RF related) Functionality has moved into the RPD. The RPD also acts as a Media Converter (Fibre to Coaxial). Dependent on Fibre topology the RPD and Node Enclosure can either displace the Conventional Analog Node or move closer to the Customer.

4. Kubernetes (K8s) Cluster Architecture

K8s is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. K8s services, support, and tools are widely available.

The following Figure shows the generic Kubernetes Cluster Architecture.

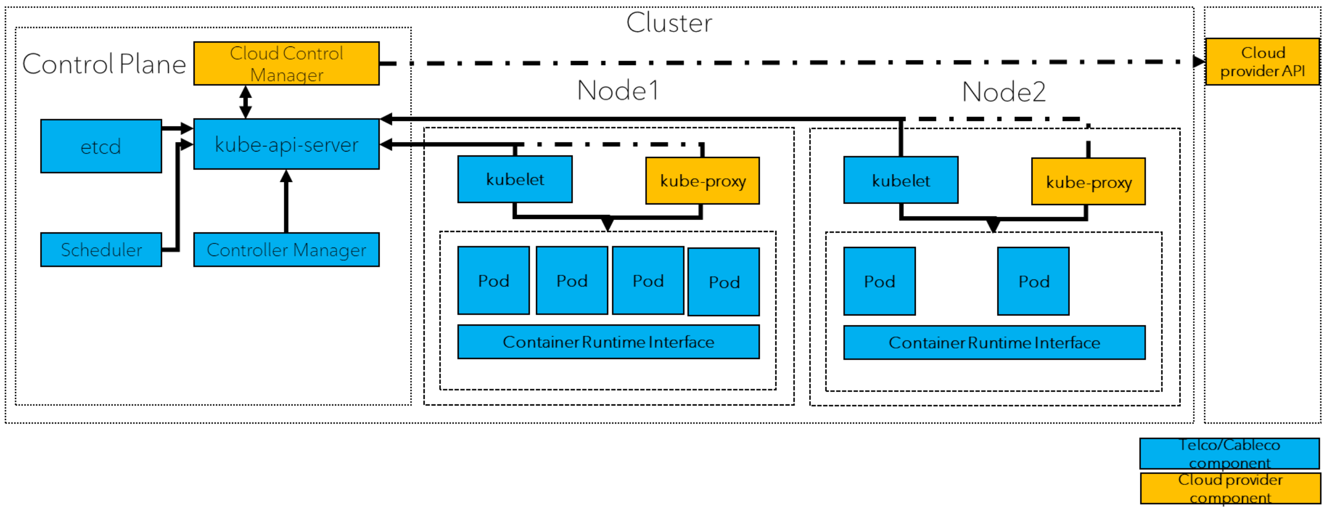


Figure 5 - Kubernetes Cluster Architecture

Unlike Virtual Machines (VMs), containers have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Like a VM, a container has its own filesystem, share of Central Processing Unit (CPU), memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers have become popular because they provide extra benefits, such as:

- Agile application creation and deployment.
- Continuous development, integration, and deployment provides for reliable and frequent container image build and deployment with quick and efficient rollbacks.
- Dev and Ops separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
- Observability: not only surfaces OS-level information and metrics, but also application health and other signals.
- Environmental consistency across development, testing, and production environment runs the same on a laptop as it does in the cloud.
- Cloud and OS distribution portability runs on Ubuntu, Red Hat Enterprise Linux (RHEL), on-premises, on major public clouds, and anywhere else.
- Application-centric management raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
- Loosely coupled, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
- Resource isolation: predictable application performance.
- Resource utilization: high efficiency and density.

Containers are a good way to bundle and run your applications. In a production environment, containers that run the applications can be managed to ensure that there is no downtime. For example, if a container goes down, another container needs to start.

K8s provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more.

K8s provides you with:

- **Service discovery and load balancing** K8s can expose a container using a Domain Name System (DNS) name or using an Internet Protocol (IP) address. If traffic to a container is high, K8s can load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** K8s allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.

- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using K8s, and it can change the actual state to the desired state at a controlled rate. For example, you can automate K8s to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.

- **Automatic bin packing** You provide K8s with a cluster of nodes that it can use to run containerized tasks. You tell K8s how much CPU and Random Access Memory (RAM) each container needs. K8s can fit containers onto your nodes to make the best use of your resources.

- **Self-healing** K8s restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.

- **Secret and configuration management** K8s lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

- **Batch execution** In addition to services, K8s can manage your batch and Continuous Integration (CI) workloads, replacing containers that fail, if desired.

- **Horizontal/Vertical scaling** Scale your application up and down with a simple command, with a User Interface (UI), or automatically based on CPU usage.

- **IPv4/IPv6 dual-stack** Allocation of IPv4 and IPv6 addresses to Pods and Services

- **Designed for extensibility** Add features to your K8s cluster without changing upstream source code.

4.1. Kubernetes Services Controller

In K8s, controllers act as control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.

A controller tracks at least one K8s resource type. These objects have a spec field that represents the desired state. The controller(s) for that resource are responsible for making the current state come closer to that desired state.

The controller might carry the action out itself; more commonly, in K8s, a controller will send messages to the Application Programming Interface (API) server.

The Job controller is an example of a K8s built-in controller. Built-in controllers manage state by interacting with the cluster (API) server.

Job is a K8s resource that runs a Pod, or several Pods, to carry out a task and then stop.

(Once scheduled, Pod objects become part of the desired state for a kubelet).

When the Job controller sees a new task, it makes sure that, somewhere in your cluster, the kubelets on a set of Nodes are running the right number of Pods to get the work done. The Job controller does not run any Pods or containers itself. Instead, the Job controller tells the API server to create or remove Pods. Other components in the control plane act on the new information (there are new Pods to schedule and run), and eventually the work is done.

After you create a new Job, the desired state is for that Job to be completed. The Job controller makes the current state for that Job be nearer to your desired state: creating Pods that do the work you wanted for that Job, so that the Job is closer to completion.

Controllers also update the objects that configure them. For example: once the work is done for a Job, the Job controller updates that Job object to mark it as finished.

In contrast with Job, some controllers need to make changes to things outside of your cluster.

For example, if you use a control loop to make sure there are enough Nodes in your cluster, then that controller needs something outside the current cluster to set up new Nodes when needed.

Controllers that interact with external state find their desired state from the API server, then communicate directly with an external system to bring the current state closer in line.

(There actually is a controller that horizontally scales the nodes in your cluster.)

The controller makes some changes to bring about your desired state, and then reports the current state back to your cluster's API server. Other control loops can observe that reported data and take their own actions.

K8s takes a cloud-native view of systems and can handle constant change.

Your cluster could be changing at any point as work happens and control loops automatically fix failures. This means that, potentially, your cluster never reaches a stable state.

If the controllers for your cluster are running and able to make useful changes, it doesn't matter if the overall state is stable or not.

As part of its design, K8s uses lots of controllers that each manage a particular aspect of cluster state. Most commonly, a particular control loop (controller) uses one kind of resource as its desired state and has a different kind of resource that it manages to make that desired state happen. For example, a controller for Jobs tracks Job objects (to discover new work) and Pod objects (to run the Jobs, and then to see when the work is finished). In this case something else creates the Jobs, whereas the Job controller creates Pods.

It's useful to have many simple controllers rather than one, monolithic set of control loops that are interlinked. Controllers can fail, so K8s is designed to allow for that.

Kubernetes comes with a set of built-in controllers that run inside the kube-controller-manager. These built-in controllers provide important core behaviors.

The Deployment controller and Job controller are examples of controllers that come as part of Kubernetes itself ("built-in" controllers). Kubernetes lets you run a resilient control plane, so that if any of the built-in controllers were to fail, another part of the control plane will take over the work.

The node controller is a K8s control plane component that manages various aspects of nodes.

The node controller has multiple roles in a node's life. The first is assigning a CIDR block to the node when it is registered (if CIDR assignment is turned on).

The second is keeping the node controller's internal list of nodes up to date with the list of available machines. Whenever a node is unhealthy, the node controller deletes the node from its list of nodes.

The third is monitoring the nodes' health.

By default, the node controller checks the state of each node every 5 seconds. This period can be configured using kube-controller-manager component.

In most cases, the node controller limits the eviction rate. The node eviction behavior changes when a node in each availability zone becomes unhealthy. The node controller checks what percentage of nodes in the zone are unhealthy at the same time.

The reason these policies are implemented per availability zone is because one availability zone might become partitioned from the control plane while the others remain connected. If your cluster does not span multiple availability zones, then the eviction mechanism does not take per-zone unavailability into account.

A key reason for spreading your nodes across availability zones is so that the workload can be shifted to healthy zones when one entire zone goes down. Therefore, if all nodes in a zone are unhealthy, then the node controller evicts at the normal rate. The corner case is when all zones are completely unhealthy (none of the nodes in the cluster are healthy). In such a case, the node controller assumes that there is some problem with connectivity between the control plane and the nodes and doesn't perform any evictions. (If there has been an outage and some nodes reappear, the node controller does evict pods from the remaining nodes that are unhealthy or unreachable).

The node controller is also responsible for evicting pods running on nodes with problems unless those pods can tolerate that taint. The node controller also adds taints corresponding to node problems like node unreachable or not ready. This means that the scheduler won't place Pods onto unhealthy nodes.

4.2. Kubernetes Worker Nodes

K8s runs your workload by placing containers into Pods to run on *Nodes*. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods.

Typically, you have several nodes in a cluster.

The components on a node include the kubelet, a container runtime, and the kube-proxy.

There are two main ways to have Nodes added to the API server:

1. The kubelet on a node self-register to the control plane
2. Manual addition of a Node object

After you create a Node object, or the kubelet on a node self-register, the control plane checks whether the new Node object is valid.

The name of a Node object must be a valid DNS subdomain name.

The name identifies a Node. Two Nodes cannot have the same name at the same time. K8s also assumes that a resource with the same name is the same object. In the case of a Node, it is implicitly assumed that an instance using the same name will have the same state (e.g. network settings, root disk contents) and attributes like node labels. This may lead to inconsistencies if an instance was modified without changing its name. If the Node needs to be replaced or updated significantly, the existing Node object needs to be removed from API server first and re-added after the update.

Node objects track information about the Node's resource capacity: for example, the amount of memory available and the number of CPUs. Nodes that self-register report their capacity during registration. If you manually add a Node, then you need to set the node's capacity information when you add it.

The K8s scheduler ensures that there are enough resources for all the Pods on a Node. The scheduler checks that the sum of the requests of containers on the node is no greater than the node's capacity. That sum of requests includes all containers managed by the kubelet, but excludes any containers started directly by the container runtime and excludes any processes running outside of the kubelet's control.

4.3. Cloud Control Manager (Informational)

Running CNF over VM had also been contemplated as a Deployment Model. Refer to Figure 9 - Wireless Core Transition path. The thought process here was that a Common Cloud Environment could be used for Resource Sharing between Wireless/Wireline workloads. Use of a Common Hardware Platform was also explored. A decision was taken to defer exploration of this framework. Refer to section 9.5 for an explanation.

5. Container as a Service (CaaS)

The CaaS architecture is typically composed of multiple layers. The following Figure shows the layers.

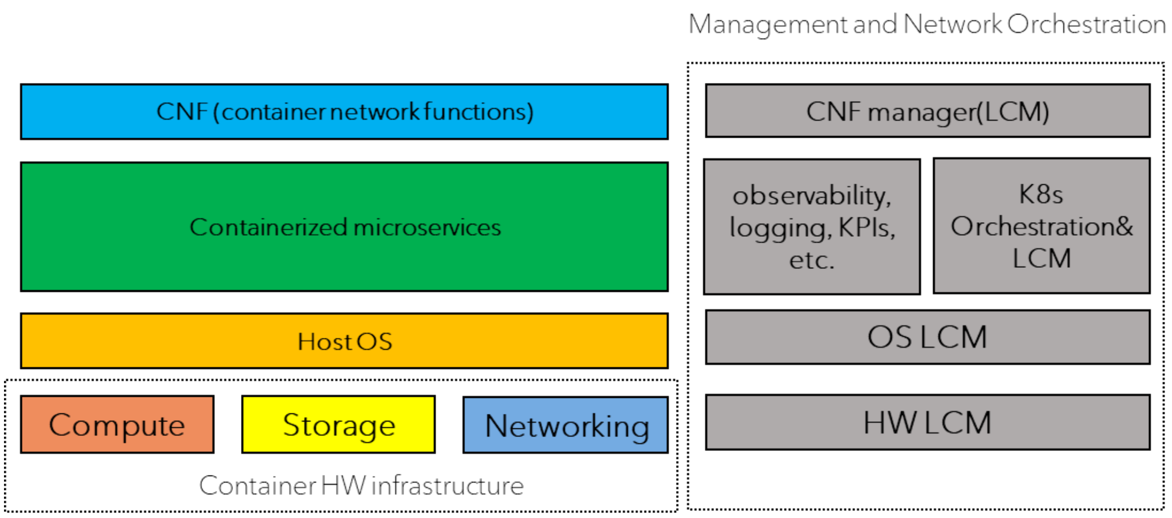

Management and Network Orchestration
CNF (container network functions)
Containerized microservices
Host OS
Compute
Storage
Networking
Container HW infrastructure
CNF manager(LCM)
observability, logging, KPIs, etc.
K8s Orchestration& LCM
OS LCM
HW LCM

Figure 6 - CaaS Architecture

1. **Infrastructure layer:** This layer encapsulates physical or virtual resources that make up the CaaS platform; for example, compute, storage, and networking resources. The CaaS provider manages all these resources, not the CaaS user.

2. **Container orchestration layer:** The second layer is responsible for container lifecycle management, which includes the provisioning, scaling, and scheduling of containers. The layer supports container orchestration tools such as K8s.

3. **Containerization layer:** This layer packages applications and dependencies into lightweight and portable containers. Such containers can be created using containerization tools such as Docker. Moreover, containers can be stored and distributed using container registries such as Docker Hub that hold container images.

4. **Platform services layer:** The fourth layer describes additional services essential for the deployment and management of containers, such as load balancing, service discovery, and logging. These services are generally offered by CaaS providers and can be accessed through APIs or web consoles.

5. **Application layer:** This is the last layer in the CaaS framework. It contains containerized applications deployed across the CaaS platform. Applications are developed using a variety of programming languages and frameworks and are later packaged as Docker images to deploy them on the CaaS platform.

The CaaS architecture offers multiple benefits to containerized applications as it simplifies the deployment and management of applications and improves the agility, scalability, and reliability of these applications. Since CaaS abstracts away container orchestration and infrastructure management, developers can focus on developing and deploying applications. Further, CaaS ("On-Premises" or Cloud) providers can manage the underlying infrastructure and operational tasks.

6. CaaS & Application Combined

The initial option chosen, to maintain continuity of current processes, was to use the Vendor as provider for:

1. CaaS Platform and
2. vCCAP Application
3. vCCAP Application Lifecycle Management (analogous to Element Management System)

For all intents and purposes, the "look and feel" of this was like the Vendor provided Appliance. This ensured that Vendor remained as the last resort to fix problems/issues. It also ensured that there was no "finger-pointing" between Infrastructure, Orchestration and Platform Services Layer Provider and Application Layer Provider.

The disadvantage of this arrangement is that the Vendor must consider all aspects of the CaaS Architecture and does not only concentrate on the vCCAP Application development.

7. (vCCAP) Deployment Model 1

This deployment model follows the Appliance based framework closely. The appliances' network modules are containerized. The containers and physical servers that they run on are representations of the underlying Hardware in the appliance.

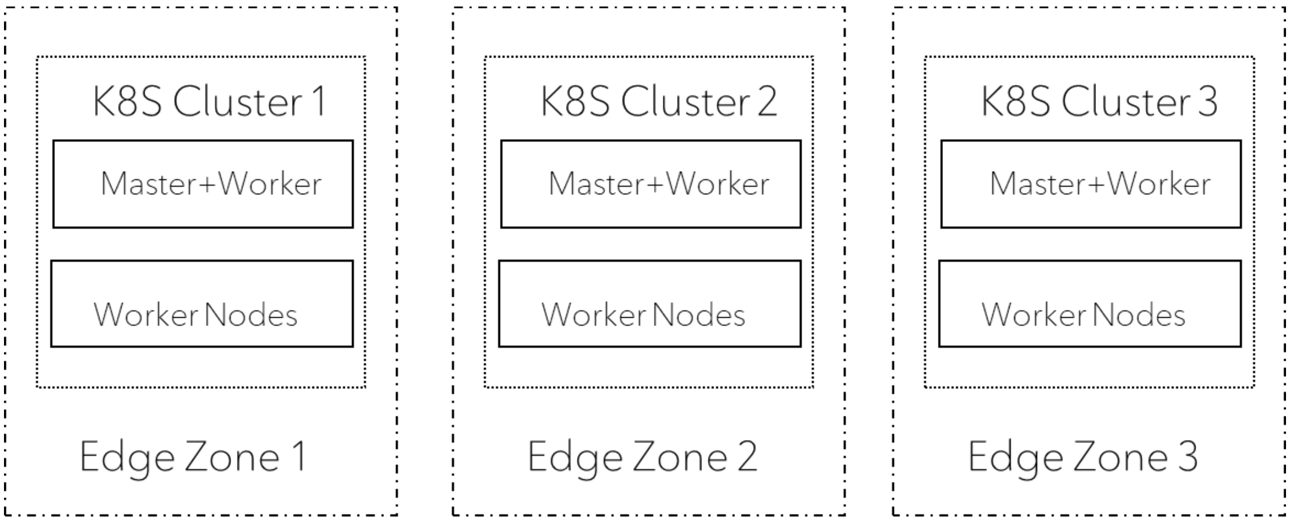


Figure 7 - K8s/vCCAP Deployment Model 1

7.1. Linecard functions in a Container

The supervisor/application controller module and the RPD Controller module are instantiated in containers running on their own physical servers. Each module from the Appliance corresponds to a container running on a physical server. The Physical Network Function (PNF) corresponds exactly to the Container Network Function (CNF)

7.2. Kubernetes Cluster View

The K8s Controller functionality co-exists with (vCMTS) Application Controller Module. The Host Agents run on each container/physical server. In this model two instances of K8s Controller run on separate containers/servers. This enables intra-cluster K8s Controller Redundancy.

8. (vCCAP) Deployment Model 2

This deployment model follows the micro-services-based framework closely. The (vCCAP) RPD-Controllers are broken up, each in their own container. Other functions are grouped together and run in different containers.

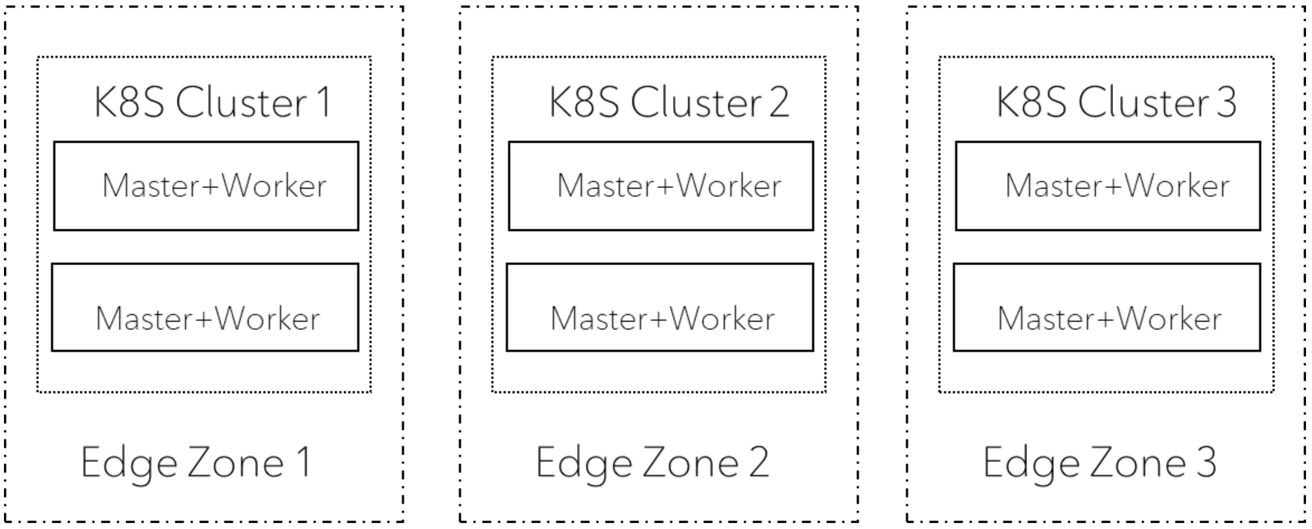


Figure 8 - K8s/vCCAP Deployment Model 2

8.1. Mixing it up (Application/Kubernetes System)

The K8s controller, host-agents and Application containers all run on the same Node. As per current scaling, this would closely model an Edge-based Deployment.

9. (Wireline) Considerations/Learnings

There are several differences in the Commissioning/Activation/Management processes in the transition from Appliance based vCCAP to Cloud-Native vCCAP.

9.1. Separation of Platform and Application Configuration

Appliance configuration could be achieved in a few steps that did not have dependencies:

1. Setup (Network Equipment) Out-of-Band Access, define a “starting” Configuration that allowed for connectivity into the Service-Provider Network
2. Complete the Configuration

Now, the configuration is broken up into additional stages and there are dependencies that exist in proceeding.

1. Set up (Server) Access (Ex. ILO, DRACS, BMC)
2. Install updated Server Firmware and OS
3. Define a “starting” Configuration that allows for connectivity to Internet/Service Provider Network (depending on where repositories etc. are located). Access to NTP, DNS is pre-required.
4. Complete Kubernetes/Platform Installation/Configuration (dependent on location of repositories configuration files etc…)
5. Complete Application Installation
6. Complete DOCSIS® Configuration

9.2. Automation

Transition from Script-based Configuration to Automated (Tool-Based) Configuration and on-going Configuration Management. This changed Commissioning processes that had been followed for quite a while.

9.3. Server, Platform and Application/Service Management

In addition to (Access Network) Service Management, Server (Hardware and Software) and Kubernetes Platform Management is now required. Depending on the "XaaS" Model chosen there could be multiple teams involved in the overall operations/management of the System!

There was a learning curve associated with the plethora of Vendor provided and Open-Source tools available for use.

9.4. Command Line Interface -> Programmatic Interfaces

The "look and feel" for interaction with the Appliance (use of Command Line Interface (CLI), followed by analysis of output) now changes. Command line can and still will be used. However, now it is more about the use of Programmatic Interfaces and (visual) analysis of the output/reports.

9.5. Platform/Application related considerations

With all the redundant equipment (both Servers and Network) as well as (Kubernetes) Platform environment available, one would assume that system malfunction would be drastically reduced. There is still work to be done in stabilizing this "brand-new" framework that has been introduced. The premise of this "self-healing" system is still a "work-in-progress."

Within the limited Field Trial Deployments that were completed, the following observations were made, and issues were encountered:

1. Intra-Cluster communication problem.
2. Reset/Reboot of entire Cluster.
3. Inter-(K8s) Container interaction problem.
4. Inadequate ventilation/cooling resulting in Hardware Platform failures.

10. (Potential) Future Changes to (Wireline) Deployment Model

Transition to separate K8s Controllers as per Figure 11 - Distributed K8s cluster Deployment Model. Use of this Deployment Model would dissociate the K8s Controllers from the Worker Nodes for additional resiliency. A requirement to use this Model would be a very resilient Core Network implementation.

11. Wireless Network Functions Virtualization

11.1. Virtualization Background

Virtualization of network functions has come a long way since the concept was adapted from virtualization of third-party independent software vendors' cloud-based workloads. AT&T domain 2.0 vision white paper published in 2013 was also quite inspirational for Telco world in a move towards virtualizations of network functions. While the hype has been around for a while, the fact is that it took a long time for Telco and cable operators to get onboard with the idea due to various challenges. The main challenge has been dealing with multiple suppliers and splitting responsibilities of hardware, software, and orchestration layer. It would not be a stretch to say that most of the challenges have been organizational and cultural rather than technical.

The other main concern around network function virtualization has been around packet processing capabilities and the scale needed to deal with high-capacity user plane functions. However, those concerns have been dealt with using technologies like smart NICs, SRIOV and DPDK. That is why it was natural for operators to start with virtualizations of control plane or signaling functions (whether it is for IP Multimedia Subsystem (IMS) or 4G core control plane) and move on to user plane subsequently.

In many cases, operators were forced to take virtualization path because of end of life and support of Physical network functions (PNFs) and network suppliers discontinuing their PNFs for certain control plane functions.

In terms of virtualization journey, early software releases by network suppliers were virtualization of equivalent PNF into monolithic software not taking advantage of underlying software modularity provided by virtualization and micro services architecture. Network vendors transformed their existing PNFs mapping one to one hardware line cards into equitant software functionality. This was to claim the availability of virtualization software by network suppliers without conforming to principles of distributed software and cloud native micro services architecture.

After initial deployments of VNFs, network operators and industry were eager to follow on to containerization of network functions or container network functions (CNFs). Some operators leap frogged the VNFs completely to go directly to CNFs. Containerization of software promised the scale, agility, and efficiencies not to be matched by VNFs hosted on virtual machines. While there are still some benefits of VNFs or software over VMs like complete isolation, security, and better manageability over containers, momentum has clearly shifted towards containers because of compute efficiencies, quick turn up time of pods versus VM and container orchestration becoming more mature. However, early deployment of CNFs was over virtual machines provided by network vendors with their own cloud infrastructure environment and CNFs.

Bare-metal CNFs availability by network suppliers followed along quickly although some vendors were supporting it from day one.

Evolution of wireless from 4G/LTE into 5G has built on the virtualization momentum and expedited the wireless operator's journey towards cloud native software deployment. This is because 3GPP standardization of 5G core networks kept cloudification of the software goal in their mind while developing standards. 5G protocols were completely transformed from legacy Telco protocols (like diameter etc.) to HTTP based making it well positioned from network suppliers to produce cloud native 5G core software and harness the benefits like web scale and software agility.

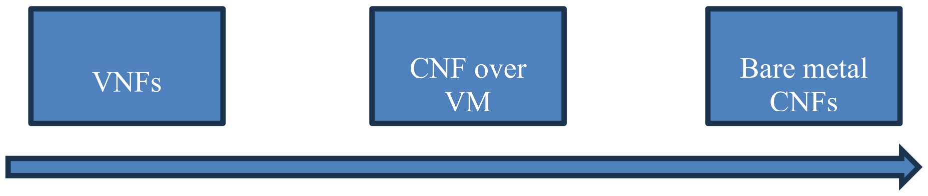


Figure 9 - Wireless Core Transition path.

11.2. Wireless Core deployment Models

From the 5G core deployment model perspective, there are two main geographical deployment models which may result in different Kubernetes control plane or cluster deployment models.

5G core functions can be broken down into control plane functions, subscriber data and user plane functions. Geographical placement of control plane functions (e.g. AMF, SMF, NSSF etc.) and subscriber data (UDM, UDR) is usually centralized in regions or nationally as delay for session set up and control can be met by centralized deployment. Also, the scale of control plane traffic is different than user plane functions. User plane functions may need lower latency depending upon application use case and need to be more distributed and closer to user. It is assumed that the reader is familiar with 3GPP defined 5G core functions and procedures.

We present two sets of deployment models for Kubernetes cluster in the following sections. These models are based on how wireless core functions can be deployed.

11.2.1. Centralized Deployment Model

Historically speaking, wireless traffic (both control and user plane) from access network is routed to few national/regional data centres hosting physical network functions of wireless core network. With the introduction of control and user plane separation (CUPS) in LTE and natively supported in 5G, it has become possible to break apart control and user plane and push user plane functions closer to edge as per use case requirement.

Despite the availability of CUPS, early deployments of 5G core functions are expected to be centralized in regional data centres. This is to serve eMBB (evolved mobile broadband) also known as internet traffic. Internet peering availability at regional or central sites is another reason to centralize even user plane functions in key regional data centres.

We can break down 5G core workloads into categories as follows.

1. Subscriber data Management (UDM, UDR)
2. Control Plane functions (AMF, SMF, NSSF, NRF, AUSF, PCF, BSF, SCP)
3. User plane functions (UPF)
4. Management and network orchestration

Subscriber database and control plane functions are very much centralized while user plane can be centralized or deployed at edge. However early deployments of cloud native user plane functions are centralized and would evolve into edge deployment as need for low latency use cases arise.

For centralized deployment in regional or national data centres, scaling requirement for CNFs is lot higher than user plane only CNFs deployments for 5G edge use cases. In this case, the incremental requirement to deploy Kubernetes control plane (aka Master nodes) on dedicated nodes is trivial. Typical requirement to deploy K8s control plane is 3 servers with high availability.

In centralized deployment model, Kubernetes clusters are deployed in regional data centres consisting of multiple availability zones or failure domains. Availability zone is defined as a data center location where compute, storage and networking resources are sharing space and fed from the same power source. We use word failure domain interchangeably with availability zone. Any power failure or local disaster like flooding would impact all resources deployed within single failure domain or availability zone. Availability zones mentioned here are not to be confused with AWS Public cloud AZ terminology although concept is similar but, in this context, operators owned on-premises data centers are being referenced. Multiple Availability zones or failure domains are contained in geographic regions and wireless traffic coming from local RAN (radio access network) is typically contained within each region.

For centralized deployment of Wireless Core functions, each Kubernetes cluster is contained in one availability zone or failure domain. Wireless core functions or CNFs are contained within one K8S cluster. For high availability purposes, CNFs or wireless core network functions are backed up from cluster in a same availability zone or from different zone in case of complete failure (geo redundancy) in one availability zone. K8S control plane is deployed with high availability 3 node configuration in each cluster. One should keep in mind that 3GPP procedures defined for service discovery and gateway selection are available at a higher layer than underlying Kubernetes pod life cycle management and service exposure. So, if a CNF available from one cluster becomes unavailable, the CNFs from other cluster should be able to back it up using native procedures defined within 3GPP for service discovery and node selection.

Also, it may be needed to have multiple Kubernetes clusters deployed in single availability zones whether due to need of using different clusters based on operational structure of organization, scaling of clusters or security reasons. The number of k8s clusters deployed in single availability zone would depend upon CNF workload requirements. Management and network orchestration workloads can be deployed as part of CNFs cluster or as a separate cluster.

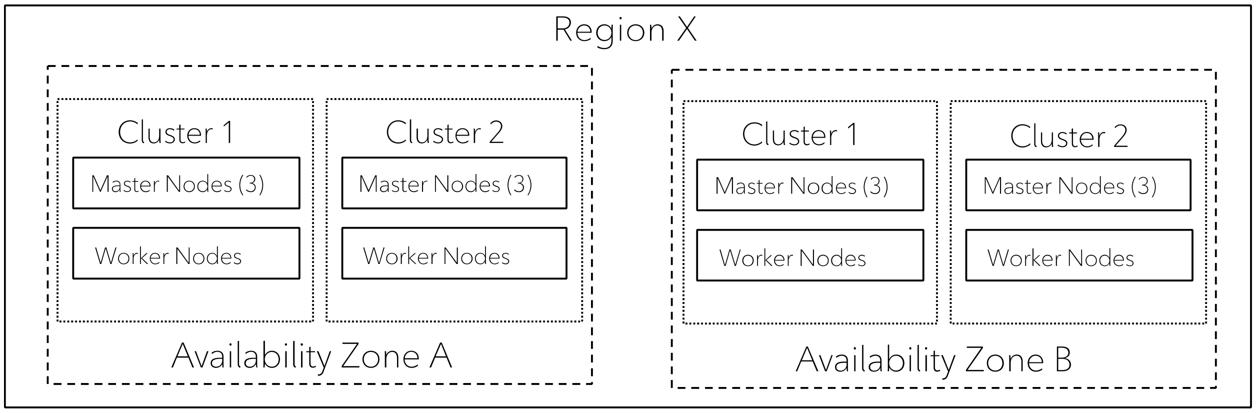


Figure 10 - Wireless Core Centralized K8s Deployment Model

The figure shows two availability zones in a region offering geo redundancy. Each availability zone has two K8s clusters available for 5G core CNFs. Each CNF and its associated micro services are contained within its own cluster with 3GPP defined procedures used for service discovery and node selection.

It is not the author's intention to discuss all possible k8s cluster deployment models for centralized deployment of 5GC functions. Other models can be considered based on network operator specific requirements.

11.2.2. Edge Deployment

Edge deployment model for 5G core is typically for distributed user plane function (UPF) or edge breakout scenario where traffic from LTE/5G access network is desired to route to the closest UPF or PGW to reduce latency. For low latency use cases, (<10 msec) typically one would deploy third party application hosted on MEC (Multi access edge compute) in same premise as network user plane functions. Edge location could be in stadium, on customer premises, or in cell tower location. We defined edge zone as a facility where wireless core functions (typically 5G user plane functions) are deployed closer to network edge.

We present three different K8s deployment models here for edge deployment of 5G core functions (where UPF is the most common one)

1. Deployment Model 1- K8S control plane in centralized sites (AZ) with user plane (worker nodes) in different edge zones- Distributed cluster across zones
2. Deployment Model 2-K8s cluster (control plane and user plane) contained in each edge zone.
3. Deployment Model 3-K8s control plane and user plane sharing nodes in each edge zone.

11.2.2.1. Deployment Model 1- (Distributed K8S cluster)

In this K8S deployment model, K8S control plane is deployed in 3 different centralized sites or availability zones for high availability. K8S user plane or worker nodes are deployed to offer CNFs in each edge zone. With this model, there is a single cluster to manage for distributed 5G workloads across different edge zones. CNF or network functions need to be contained in each edge zone to offer low latency benefits from 5G user plane function (UPF) perspective. K8S cluster is split here across different data centres. This model may complicate cluster networking distributed across different edge zones. As k8s control plane is controlling multiple edge zone worker nodes, k8s control plane is deployed with high availability and geo redundancy.

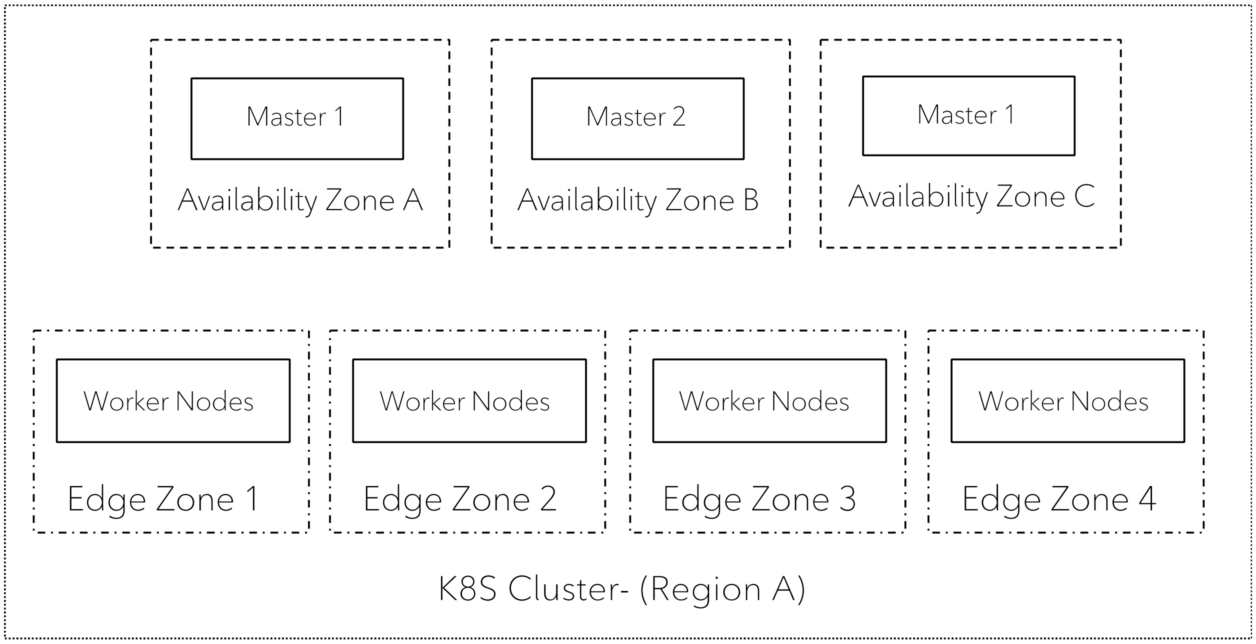


Figure 11 - Distributed K8s cluster Deployment Model

11.2.2.2. *Deployment Model 2*

In this deployment model, K8S control plane and user plane is deployed in same edge zone and k8s cluster is contained in its own edge zone. Depending upon the criticality of the workloads, master nodes can be locally redundant (2 or 3) to provide high availability. However, there is no geo redundancy available for K8S control plane but since the cluster is contained in single edge zone, both control plane and user plane are sharing fate. In case of losing control plane, only cluster in the edge zone is impacted. This model is similar to the model described in section 11.2.1other than the fact that cluster size is small, and it is used for 5G core edge workloads (e.g. distributed UPF). This model would increase the number of clusters to manage and need more resources to duplicate K8s control plane in each edge zone but offers simplification with contained networking across k8s cluster.

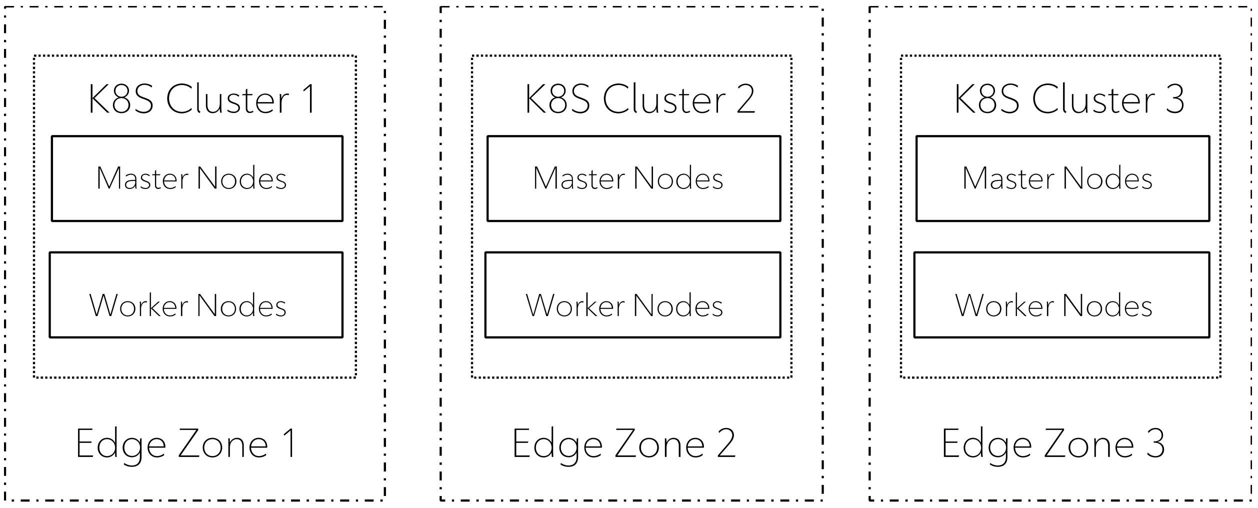


Figure 12 - K8s Cluster Edge Deployment Model

11.2.2.3. *Deployment Model 3*

In this deployment model, K8s cluster is contained within edge zone but no dedicated nodes for k8s control plane. Control plane and user plane share the same nodes.

Edge deployment for 5G user plane function is typically in sites which have limited real estate and power. In these scenarios with limited space and power, Kubernetes control plane and user plane nodes can be shared to optimize compute. Number of nodes can be scaled to meet workload requirement. A lighter version of Kubernetes K3s can also be deployed in these situations. Bare metal deployment of containers is ideal in such circumstances to save compute resources. Some CNFs software vendors may also package their solution with HW provided by them in a model like network appliance.

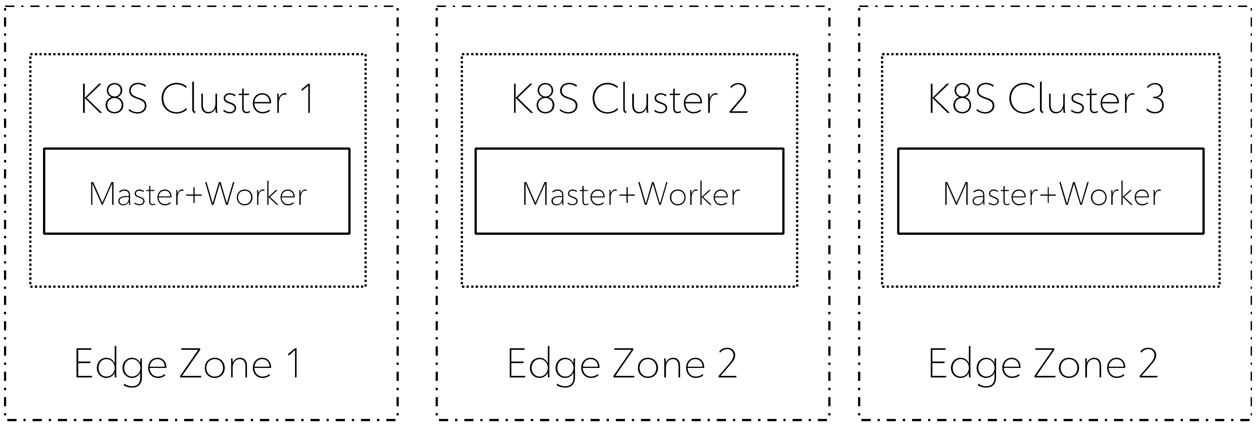


Figure 13 - K8s Cluster compact Deployment Model

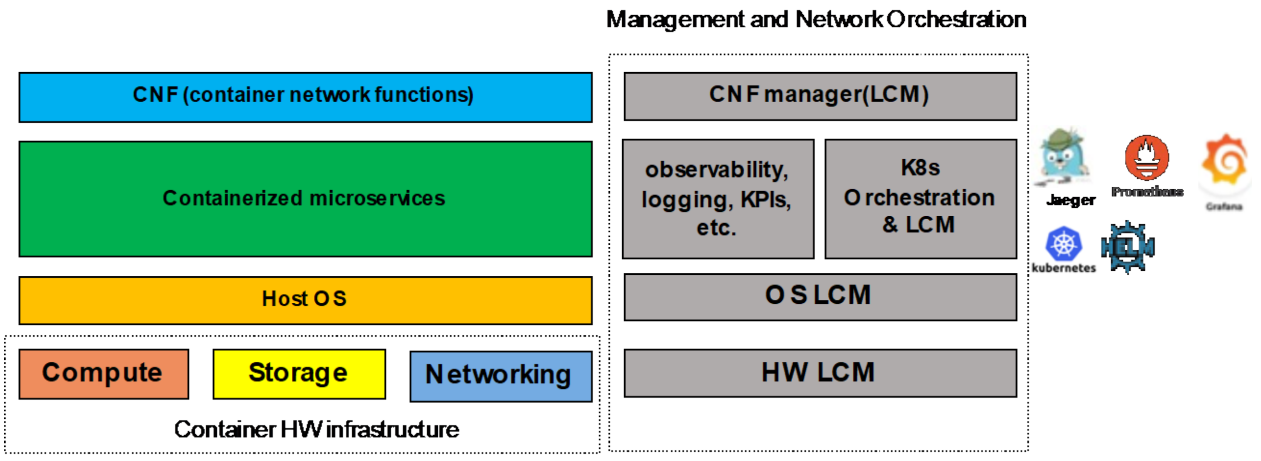
11.3. Life Cycle Managemnt

Containerization or virtualization of network functions disaggregates the software and hardware making it possible to use common off the shelf hardware supported by VNFs and CNFs vendors. However, it comes with the complexity of dealing with separate vendors for HW infrastructure, host OS, network vendors, orchestration and interoperability needed for them to work together. This has been a significant shift from the world of PNFs (Physical Network functions) where a single vendor provided Hardware and Software bundled together and a single team within the network operator was responsible for managing it.

We refer to the compute, storage, and networking resources as HW infrastructure needed to run containers and containers network functions.

The following Figure shows the main layers involved in building cloud native infrastructure and main options for management and network orchestration.

Usually, CNF supplier provides the cloud resource requirements (compute, network throughput etc.) and other KPIs for their CNF to run optimally. CNF provider typically provides CNF manager for life cycle management and orchestration of container network function. If there are multiple CNF suppliers, then there would be multiple CNF managers provided by each network vendor. CNF manager provides a similar function to what VNF manager does as defined by ETSI MANO architecture framework for NFV. Then the underlying layer is containerization layer and container orchestration is provided by plain vanilla k8s or CNF vendor provided Kubernetes. Network operators can run and manage their own Kubernetes layer, rely on CNF provided k8s or third party CaaS platform provider such as RedHat OpenShift. Container observability, logging and tracing can be achieved by operator owned open-source tools like Grafana, Jaegar and Prometheus or third part providers. Many third-party CaaS (container as a service)


Management and Network Orchestration
CNF (container network functions)
Containerized microservices
Host OS
Compute
Storage
Networking
Container HW infrastructure
CNF manager(LCM)
observability, logging, KPIs, etc.
K8s Orchestration & LCM
OS LCM
HW LCM
Jaeger
Prometheus
Grafana
kubernetes

Platform providers package opensource tools in their solution by validating with certain Kubernetes version and providing support to operator.

Since containers are running on the host operating system, host OS system dependencies and life cycle need to be managed. Then the lowest layer is hardware infrastructure including compute, storage and networking resources needed for containers.

While container infrastructure and CNFs can be procured from different vendors, a single vendor can provide management and network orchestration capabilities for the whole solution. If there is a single CNF supplier, it can validate and certify all the cloud infrastructure components with its CNFs and provide the golden template to operator to build infrastructure accordingly. Infrastructure and container orchestration then can be taken care of by CNF vendor provided management and orchestration solution. This can avoid many issues associated with interoperability, integrating eco system players and vendor finger pointing at each other.

The other option can be where Operator builds its own MANO tooling and HW infrastructure using open-source software or rely on third party CaaS platform provider like Redhat OpenShift. This approach has its own advantages if a network operator has multiple CNF vendors or cloud infrastructure that is common across IT and network leveraging common tooling for different container workloads. However, this approach needs more effort and time for integrating different vendor solutions and finger pointing if CNFs' KPI are not met.

12. Conclusion

In this document, the Authors have shown the transitions made from Appliance Based Systems to Cloud-Native based Frameworks for both Wireline Access as well as Mobile/Wireless Core Networks. The initial steps to make the transition have been taken. The initial steps taken have put the Vendor (of the original Appliance Based Equipment) in charge of the entire CaaS Architecture / Layers:

1. Server Hardware
2. Server Software
3. Container Orchestration
4. Platform Services
5. Application (CNF)
6. Platform & Application Management

The next steps would include:

1. Stabilization of the Platform Services
2. Stabilization of the Application Delivery and Management

For the wireline infrastructure deployment, this could then lead to:

1. Extending the CIN to allow for RPDs to “home-in” to multiple locations.
2. Further Disaggregation of the Kubernetes Control Plane
3. Extending the (vCCAP) Applications to be versatile enough to be moved to multiple locations.

Only then could we achieve a faster recovery from service disruptions. So, the answer to the question: “Are we there yet” would be: “The Journey continues”!

Abbreviations

3GPP	3rd Generation Partnership Program
5G	5th Generation
AMF	Access and Mobility Management Function
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AUSF	Authentication Server Function
AZ	Availability Zone
BSF	Binding Server Function
bps	bits per second
CaaS	Container as a Service
CCAP	Converged Cable Access Platform
CI	Continuous Integration
CIN	Converged Interconnect Network
CLI	Command Line Interface
CNF	Container Network Function
CPU	Central Processing Unit
CUPS	Control and User Plane Separation
DNS	Domain Name System
DSI	Downstream Side Interface
FEC	forward error correction
FPGA	Field Programmable Gate Array
GPU	General Purpose Unit
HD	high definition
HW	Hardware
IP	Internet Protocol
K8s	Kubernetes
LCM	Life Cycle Management
LTE	Long Term Evolution
MANO	Management and Orchestration
NRF	Network Repository Function
NSI	Network Side Interface
NSSF	Network Slice Selection Function
OS	Operating System
PCF	Policy Control Function
PNF	Physical Network Function
RAM	Random Access Memory
RF	Radio Frequency
RHEL	Red Hat Enterprise Linux
RPD	Remote PHY Device
SCTE	Society of Cable Telecommunications Engineers
SCP	Service Communication Proxy
SL	Spine Leaf
SMF	Session Management Function
SW	Software
UDM	User Data Management
UDR	User Data Repository

UI	User Interface
UPF	User Plane Function
VM	Virtual Machine
VNF	Virtual Network Function

Bibliography & References

1. https://kubernetes.io/docs/concepts
2. https://www.spiceworks.com/tech/devops/articles/what-is-caas
3. https://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf
4. https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf