TELECOM INFRA PROJECT

Open Core Network Project Group

Cloud-Native Infrastructure

Technical Requirements v1.0

Authors:

Rabi Abdel , Vodafone

- abdel.rabi@vodafone.com

Boris Renski, FreedomFi

- brenski@freedomfi.com

Contributors:

Nick Chase, Mirantis

- nchase@mirantis.com

Joshua Braeger, Facebook Connectivity

- jbraeg@fb.com

Editor:

Chris Morton, Isn't That Write

- chris@isntthatwrite.com

Change Tracking

Date	Revision	Author(s)	Comment
March 15, 2020	v0.1	Boris Renski	Initial Table of Content
March 20, 2020	V0.1a	Nick Chase	Overview Section
May 20, 2020	V0.1b	Rabi Abdel	Requirements
May 25, 2020	v0.1c	Boris Renski	Implementation Specification
May 28, 2020	v0.1d	Boris Renski	Public Deployment / Far Edge
June 11, 2020	v1.0	Boris Renski / Stephani Fillmon	Edits for Readability
Sept. 22, 2020	v1.01	Chris Morton	Cleaned up formatting. Proofreading and line editing. Minor editorial revising. Checked conformance with (US) Plain Writing Act of 2010. Graphics manipulation and captioning.

Table of Contents

Table of Figures

No table of figures entries found.

Chapter 1 – Introduction

The purpose of the Open Core Network (OCN) project is to build a cloud-native converged packet core. As a cloud-native system, OCN will consist of a containerized microservices architecture. This document provides requirements for the portable, cloud-native infrastructure substrate on which that architecture will run.

By providing a specific substrate architecture, OCN makes it possible to create a cloud-native infrastructure of microservices that is portable across environments, thereby enabling operators to run components in various locations and situations—from on-premise commodity hardware to public cloud or even a combination of both.

1.1 Goal

This document describes the best-of-breed infrastructure to deploy and run the microservices that comprise the OCN. It serves the following purposes:

- To serve as a single source of truth defining the architecture to be used by all participants of the OCN development process— creating an environment on which to develop, deploy, and run the microservices that comprise OCN
- To specify the Telecom Infra Project (TIP) lab staging environment to use for testing and field proof-of-concepts (POCs) of various deployment OCN topologies
- To enable operators interested in conducting OCN field trials to obtain an understanding of the type of infrastructure they need to set up and budget in their respective staging and production environments

1.2 Document Scope

The purpose of this document is to describe requirements for cloud-native infrastructure for running OCN components on an individual Kubernetes (K8S) cluster, including:

- Hardware bill of materials (BOM)
- Networking setup
- Considerations for running OCN services in the public cloud
- Details and configuration information for a containerized, cloud-native substrate based on Kubernetes

This document does not include:

- Configuration or setup for networking between multiple Kubernetes clusters, as multi-cluster networking is use-case specific
- A description of the CI/CD infrastructure to be used in the development of core OCN microservices, which will be covered in a separate document
- Requirements for an orchestrator that would manage the OCN microservices themselves. This document ends at defining the northbound infrastructure APIs and assumes that

any operations pertaining to managing the microservices, such as placement decisions and updates, are delegated to the orchestrator as defined by OCN Workstream 2

- Lifecycle management tooling for cloud-native infrastructure. This task will be handled by a vendor of the operator's choice

1.3 Requirements Approach

Portability across environments is essential to OCN success, which also equates to standards adherence. Therefore, the approach we took closely collaborates with a number of standards bodies and open source communities such as the Cloud iNfrastructure Telco Taskforce (CNTT) and LF Edge Akraino stack.

The described infrastructure is fully conformant with CNTT Reference Architecture 2 (RA2), CNTT Edge, and LF Edge Akraino blueprints. It's also vendor-neutral, so operators aren't tied to a any one vendor. Any operator or vendor can implement this architecture with commodity or public cloud hardware and best-of-breed, open source software components.

1.4 Document Overview

This document includes the following chapters:

- Overview of infrastructure standards bodies

  Provides a brief overview of prominent standards bodies that do work to define a common, reusable, cloud-native infrastructure for operator deployments. These groups include CNTT, Linux Foundation Networking (LFN), Cloud Native Computing Foundation (CNCF), and Open Platform for Network Functions Virtualization (OPNFV). It focuses on CNTT and how and why it's relevant to OCN.

- What is cloud-native for a telecom operator

  Defines *cloud-native* and explains various documents produced by CNTT; in turn these lead to the Kubernetes-based reference architecture that forms the basis of this document.

- Requirements for cloud-native OCN infrastructure

  Provides an overview of various components and requirements of the cloud-native infrastructure underlay, as defined by various CNTT telecom operators.

- Kubernetes implementation requirements

  Provides a summary and rationalization of component-level architecture, as well as specific implementation choices used to implement OCN in TIP labs—including server types and networking architecture. Describes relevant CNTT requirements and how to fulfill them, and also how to create a highly available (HA) architecture for OCN.

- Public cloud deployment considerations

  Describes special considerations for deploying and running Kubernetes in the public cloud vs. a private telecom operator datacenter, such that no dependency on any one public cloud vendor is created.

- Running OCN components at the edge

  Discusses proper approaches to infrastructure automation for use cases where OCN is deployed in a geo-distributed, multi-site topology where some of the components are running in the far edge as a single server or appliance.

Chapter 2 – Overview of Infrastructure Standards Bodies

Multiple standards bodies cover the field in which OCN is working. They work to make infrastructure and workloads portable—or at least interoperable—across environments. For this reason, operators generally closely collaborate with these bodies to ensure that the standards they produce are both appropriate and helpful. Standards bodies governing the OCN arena include:

- CNCF Telco User Group (TUG) – Defines cloud-native network function (CNF) requirements and principles to which CNFs are expected to adhere. CNCF TUG is a governance group that oversees and influences the development efforts of the CNF Conformance Program (a program to certify CNFs) and the CNF Testbed project.
- LF Edge Akraino stack – A set of open infrastructures and application blueprints created by the Akraino community for a variety of uses, including 5G, internet of things (IoT), and infrastructure as a service (IaaS).
- Open Project for Network Functions Virtualization (OPNFV) – Produces standard configurations and test harnesses for various use cases. OPNFV also discovers gaps in existing projects (e.g., OpenStack, OpenDaylight Project) and works with those projects to ensure that the gaps are closed.
- Cloud iNfrastructure Telco Taskforce (CNTT) – An LFN and GSMA-sponsored task force focused on minimizing the number of network function virtualization infrastructure (NFVI) configurations used in telco deployments. With reduced variability in NFVIs, network operators can expect to reduce testing times and accelerate deployment of new capabilities. Network function providers should also appreciate economies of scale as fewer NFVI variants will be requested by operators.

2.1 CNTT overview

CNTT began as a temporary task force created by OPNFV to study requirements for telco-based use cases. But it has since grown into a much more substantial project. CNTT acts as a proxy for multiple standards bodies, creating a set of configurations and requirements that bridge the gaps required to conform to multiple standards already created by those organizations.

Considering that CNTT was largely created by telcos, it's not a surprise that many operators such as AT&T, Verizon Wireless, Orange, Telstra, and Vodafone are heavily involved, with more joining as time goes on. In addition, many vendors serving those operators, such as Ericsson, Nokia, and Huawei, as well as software vendors such as Red Hat, VMware, and Mirantis also contribute to the project.

The following image illustrates CNTT contributors, supporters, and sponsors:

Figure 1 – CNTT contributors, supporters, and sponsors

Because of such broad and widening support, we have opted to collaborate with CNTT on this requirements document, which conforms to its container-based reference architecture.

Chapter 3 – Cloud-Native Principles

From the beginning, the intention has been for OCN to be based on cloud-native principles. While the CNCF defines *cloud-native*, CNCF TUG has an overlapping definition that is more specific to telco use cases. It includes qualities such as:

- Scalability
- Declarative APIs
- Resilience
- Manageability
- Observability
- Dynamic environments
- Immutable infrastructure
- Microservices-based architecture
- Ability to create robust automation
- Use of service mesh technology to control traffic flow

Building cloud-native infrastructure by adhering to cloud-native principles is likely to result in a great variety of non-interoperable and non-portable implementations. In defining OCN cloud-native infrastructure, we stayed aligned with architecture principles described above—but more importantly, we adhered to fulfill the following objectives:

- **Enable feature velocity** – The overarching goal of the cloud-native system is to permit frequent implementation of high-impact changes with minimal effort and a high level of predictability through automation.
- **Portability** – To decrease system maintenance cost and maximize vendor independence, the cloud-native infrastructure must be portable across bare metal and cloud platforms.

OCN collaborated with CNTT to satisfy the above objectives—specifically focusing on container-based reference architecture (known as RA2). To achieve portability and vendor independence goals, CNTT defines multiple layers of documentation.

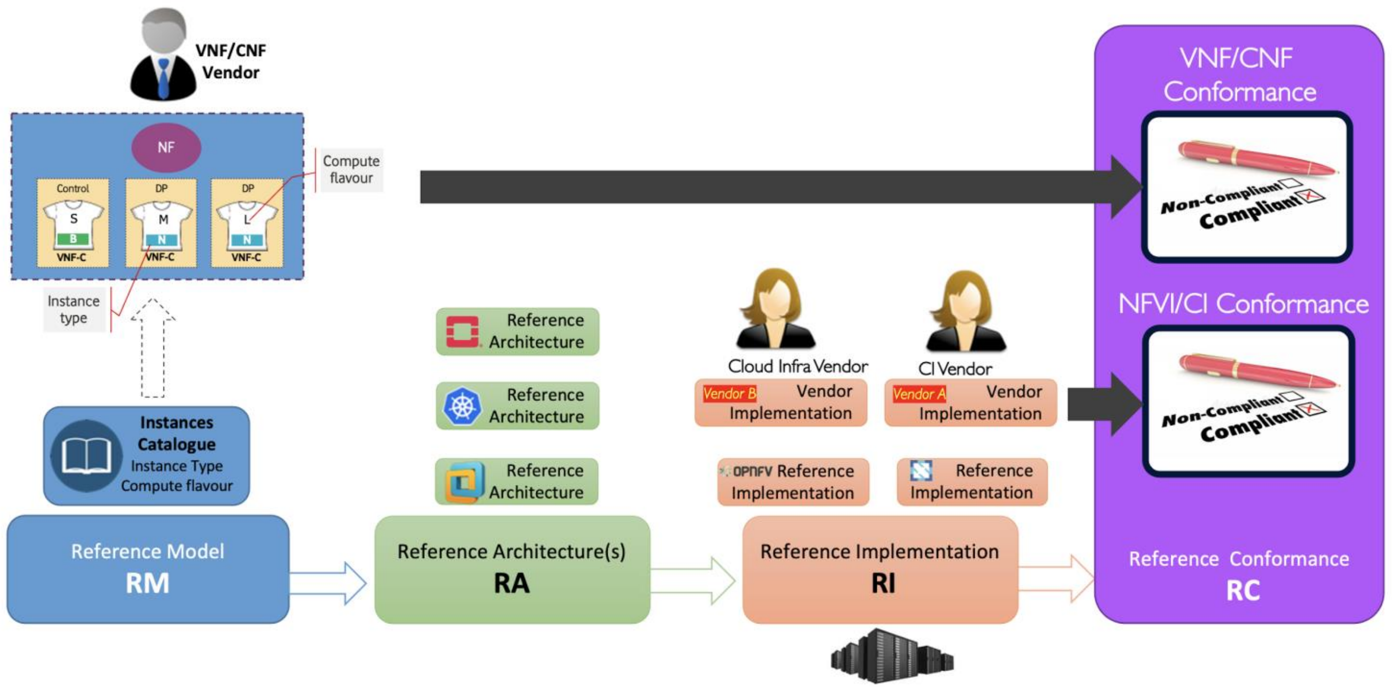

Figure 2 – CNTT documentation layers

Reference Model – CNTT is based on the *Reference Model* (RM), a technology-independent document that simply defines the qualities any system must have without specifying the means for satisfying those requirements. For example, the RM defines Crypto acceleration, network protocols, and security features, but leaves the specifics to the individual reference architectures. The RM also provides specific keys for each requirement so each can be easily referenced within other documents.

Reference Architecture – Within CNTT, a *Reference Architecture* (RA) is a specification for RM abstractions. For example, an RA might specify a certain interface, storage type, and network capability. RAs are technology-specific. The first RA is virtual machine-based and is known as RA1; the second is a container-based architecture (RA2, upon which this document is based).

Reference Implementation – Once a reference architecture is created, the community works on an implementation that satisfies those requirements. The *Reference Implementation* (RI) isn't necessarily meant to be used in production; rather it's for developing and testing workloads and validating the reference conformance suite.

Reference Conformance – The purpose of initiatives such as OCN and CNTT is to promote portability and interoperability, so it's essential to ensure that infrastructure implementations and workloads conform to the standard. As such, CNTT defines a *Reference Conformance* (RC) suite that verifies all requirements.

Our RA2 architecture is an RI example specific to OCN. RA2 is a container-based mapping of the goals and concepts of the RM to real-world system components. It's based on the Kubernetes container orchestrator—an open source platform that enables automation of containerized workloads and services.

Kubernetes enables perform configuration using both declarative and API-based methods. While other container orchestration systems such as Apache Mesos and Docker Swarm exist, CNTT and, consequently OCN, have chosen Kubernetes for the following reasons:

- Kubernetes is the broadly adopted, de facto standard for cloud-native deployments
- Developers and potential participants are most familiar with Kubernetes
- Kubernetes is the architecture many operators are gravitating toward
- OCN is meant to be portable in all environments, including private and public clouds, as well as bare metal. Kubernetes has the greatest support for this hybrid approach
- Kubernetes has a robust infrastructure of third-party support for additional hardware and software plugins

OCN has chosen to follow the guidance of operators participating in CNTT by deciding not to add additional functionality to Kubernetes. Instead it makes use of standard capabilities or existing, well-supported, community-aligned extensions if at all possible. If that is not possible, CNTT documents the requirement, then requests missing functionality from the appropriate project.

RA2 provides detailed requirements for a number of areas, including:

- Cloud infrastructure software profile capabilities
- Virtual network interface specifications
- Cloud infrastructure software profile requirements
- Cloud infrastructure hardware profile requirements
- Cloud infrastructure management requirements
- Cloud infrastructure security requirements

RA2 also provides specific requirements for the Kubernetes cluster, such as:

- Autoscaling capabilities
- Ability to run containers within a virtual machine
- Ability to integrate SDN controllers
- Support for dual-stack IPv4 and IPv6 workloads
- A highly available infrastructure
- Support for persistent storage

The next chapter lists requirements for the various aspects of an RA2-conformant architecture.

Chapter 4 – Cloud-Native OCN Infrastructure Requirements

Building an infrastructure substrate that satisfies the parameters covered in chapter 3 for operators implies following a set of operator-specific requirements. Through its collaboration with CNTT operators, OCN has outlined the following requirements and configuration parameters for the specific Kubernetes components:

- **Host OS** – Must support Linux 3.10+ for both control and worker nodes and Windows 1809 (10.0.17763) for worker nodes. The chosen operating system must be immutable, compatible with the kubeadm tool, and have no container base image restriction.
- **Version** – Kubernetes must be one of three latest minor versions (n-2)
- **Services and features** – Master nodes must run the following Kubernetes control plane services:
    - kube-apiserver
    - kube-scheduler
    - kube-controller-manager
- **Kubelet features** – The following kubelet features must be enabled:
    - CPU manager
    - Device plugin
    - Topology manager
- **etcd nodes** – Either three, five, or seven nodes running the etcd service must be running. They can be colocated on the master nodes or run on separate nodes, but cannot run on worker nodes.
- **High Availability** – Each availability zone or fault domain must run at least one master and one worker node to ensure high availability of Kubernetes control plane services and Kubernetes-managed workload resilience.
- **Separation of services using at least two nodes** – Master node services (including etcd) and worker node services (e.g., consumer workloads) must be kept separate, so there must be at least one master and one worker node.
- **Workload independence** – Workloads must not rely on availability of the master nodes for successful execution of their functionality. In other words, loss of the master nodes might affect non-functional behaviors such as healing and scaling, but components already running will continue to do so without issue.
- **Container runtime** – Container runtime might consist of one of several options, including container-d, Docker CE (via the included dockershim), and CRI-O, as long as it’s conformant with the Kubernetes Container Runtime Interface (CRI) and the Open Container Initiative (OCI) runtime specification. It must also include kernel isolation using a technology such as kata-containers.
- **CNI plugins** – Must use a multiplexer/metaplugin, such as DANM or Multus, that provides the following features:
    - The architecture must support network resiliency
    - The architecture must be fully redundant

  - The networking solution should be able to be centrally administered and configured
  - The architecture must support dual-stack IPv4 and IPv6 for Kubernetes workloads
  - The architecture must support capabilities for integrating SDN controllers
  - The architecture must support more than one networking solution
  - The architecture must support the ability for an operator to choose whether or not to deploy more than one networking solution
  - The architecture must provide a default network that implements the Kubernetes network model
  - The networking solution must not interfere with, or cause interference to, any interface or network it doesn't own
  - The architecture must support cluster-wide coordination of IP address assignment
- **Storage** – The following feature gates must be enabled: CSIDriverRegistry and CSINodeInfo. Distributed storage, specific to the environment (e.g., EBS in AWS or NetApp/EMC in private DC) can be used via volume plugin. For more information, see the CNTT chapter on container storage services.
- **K8s application package manager** – The architecture must support open APIs.

Chapter 5 – Kubernetes Implementation Requirements

Chapter 4 requirements provide certain architectural choice constraints, but don't dictate specifics of the implementation. Herein we define OCN decisions made in collaboration with an operator group regarding requirement fulfillment for each of those categories. And we provide implementation details of staging an OCN environment on bare metal servers in TIP Labs.

5.1 – Requirements for K8S Components and Configurations

Requirements Category	OCN Choice
Host OS	OCN uses one of the two latest LTS versions (n-1) of Ubuntu. Current version: 18.04
Kubernetes Version	OCN uses one of the three latest minor versions (n-2) of Kubernetes. Current version: 1.18
Master Node Services	OCN K8S master nodes run kube-apiserver, kube-scheduler, and kube-controller-manager, as well as a number of additional components to ensure high availability of K8S control plane.
Kubelet Features	OCN K8S cluster ensures the following kubelet features are enabled: CPU Manager, Device Plugin, and Topology Manager.
etcd Nodes	OCN runs a minimum of three (3) etcd nodes.

High Availability	OCN runs nodes in three regions or zones, provided the scale of a single cluster permits this (section 5.4).
Service Separation Using at Least Two Nodes	OCN deploys a six-node system that includes both control and user plane nodes.
Container Runtime	OCN uses Docker CE.
CNI Plugins	OCN uses DANM with Traefik as an ingress controller.
Storage	OCN K8s uses persistent volumes and volume plugins.
K8s Application Package Manager	OCN uses Helm 3.
Virtual Machine Manager	KubeVirt

5.2: Requirements for Physical Hardware and Bill of Materials (BOM)

To make the cloud-native requirements tangible and actionable, OCN has partnered with CNTT and TIP Labs to define the hardware BOM and network configuration for an OCN microservices staging deployment. Rather than being a hard prescription, the staging environment aims to exemplify a vendor BOM for commercial OCN pilot projects.

The choice of hardware vendor and physical network configuration can vary from operator to operator. That said, we recommend that operators deploying OCN familiarize themselves with the following requirements and attempt to stay reasonably close to them. We do not mandate full adherence to the following requirements.

K8s requirements outlined in this document don’t require that operators exclusively use bare metal. Rather, they’re intended to be “cloud and hybrid friendly.” But because we believe a large number of core network deployments will arise in operator datacenters, we paid particular attention to defining bare metal environments. Additional guidelines for deploying the same K8s architecture in the cloud are provided in Chapter 6 – Public Cloud Deployment Considerations.

Participating CNTT operators have defined two physical hardware profiles to be used in cloud-native deployments: basic and network intensive. Each uses the same requirement set, but with different values for the various CNTT RM parameters. Refer to chapter 4 of the CNTT documentation.

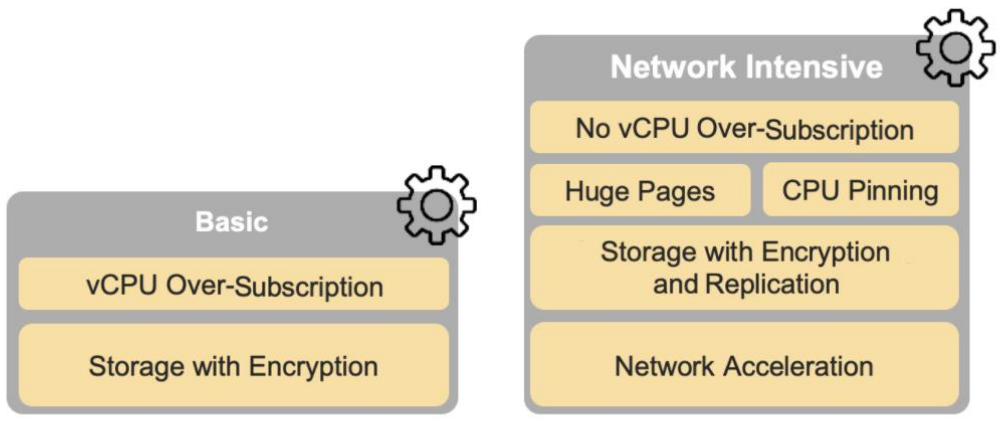

Figure 3 – Defined bare metal hardware profiles

To remain CNTT-conformant with, we've chosen a hardware BOM that adheres to the previously-described OCN parameters. But for the sake of simplicity, OCN exclusively uses the network intensive server configuration in its staging environment. And we've selected a server platform having ample room for additional networking interfaces so as to experiment with various acceleration solutions .

A CNTT-compliant K8s pod must meet the following requirements:

- One (1) physical server dedicated as a *jump* or *test* host
- Six (6) physical servers, serving as either compute or controllers
- A configured network topology permitting out-of-band management, admin, public, private, and storage networks

TIP Labs will host an OCN staging environment consisting of six cloud servers. To simplify the footprint, one of the compute servers will double as a traffic generator (it can also be used as a jump server per CNTT requirements). When deploying a similar environment in your datacenter for OCN staging, we recommend using a minimum of five physical servers (three control, or master servers, and two compute servers). If needed, you can scale the number of compute servers to whatever number that can be accommodated by datacenter rack configuration.

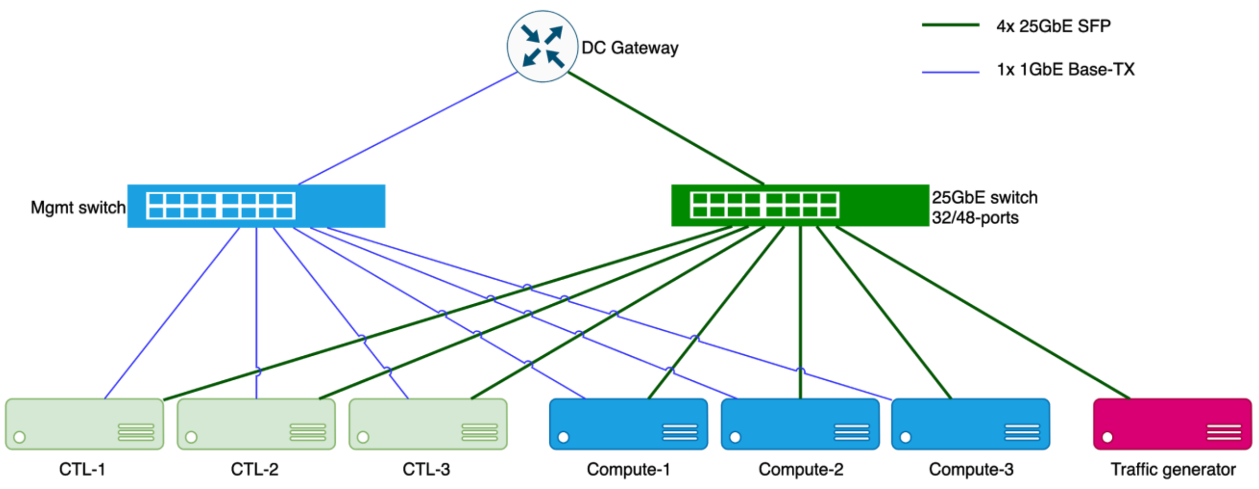

Figure 4 – CNTT-compliant Pod architecture

Per CNTT recommendation, lab hardware should stay reasonably close to the following minimum requirements:

- **CPU**
  - 2x x86_64 CPU sockets (both populated), providing 24 cores each, 48 simultaneous multi-threads (SMT) at 2.2 GHz
- **Memory**
  - 512 GB RAM
- **Storage**
  - 3.2 TB SSD via SATA 6 Gbps
- **Network interfaces**
  At least one network interface must be capable of performing PXE boot; that network must be available to serve as both the jump or test host and each physical server.
  - 4x 25 Gbps Ethernet ports, implemented as two distinct, dual-port NICs
  - Out-of-band management port
- **25 GbE switch requirements**
  - 32/48x 25GbE SFP ports
  - 6 servers x 4 ports = 24 ports + uplink
  - 7 servers x 4 ports = 28 ports + uplink
- **Management switch requirements**
  - 24x 10/100/1000 Base-TX ports

The following table describes the specific BOM for the TIP Labs OCN staging environment:

D52BQ–2U		**Control/Compute Nodes for TIP OCN Automation Project**	**Total of Six(6) Servers**
Proposed Product	**QPN**	**Specification**	**Quantity/ Server**
BASE	20S5BMA03M0	Server QuantaGrid D52BQ 2U 2.5 expander 35x40 LBG-4 (for rear NVMe)	1
CPU_CSL_SP	AJSRF9HUA00	Intel Xeon Platinum 8260,2.40G,35.75M cache, 24C/48T(165W), Max Mem 2933MHz, Turbo2.0, HT, SRF9H	2
RAM	2S5BRM320I0	DDR4-R 32G 2933MHz 1.2V	16
RAM DUMMY	EBS2S002010	DDR4 dummy DIMM	8
PSU	2S5BPWR00B0	QuantaGrid D52BQ 1*1100W -48V DC 86.3 gold grouping	2
POWER CORD	DDS5BEPB100	DC power cord for -48VDC PSU	2
RATING LABEL	HCS5B204010	QuantaGrid D52BQ 1100W PSU rating label (made in Taiwan)	1
PACKAGE	22THLPKTAE1	For vendor cons. packing	1
CPU_HS	2S5BHS00000	QuantaGrid D52BQ/D52BM CPU H/S 41F HP group	2
RAIL_KIT	FBS5B212010	QuantaGrid D52BQ/D52BM rail kit	1

ACC_KIT	25S5BAKST00	QuantaGrid D52B/D52BQ/D52BM accessory kit	1
SAS_MEZZ	1HYQZZZ032X	QuantaGrid D52BQ QS 3516 16i SAS R6 mezz (2.5) 4G	1
SATA SSD	ABS019T8001	Intel 2.5-7mm 1.92T SATAIII S4510 SSDSC2KB019T801	4
PCIE HHL CARD_R1_R2_R3ADPC0710009	ADPC0710009	NIC Intel XXV710-DA2 PCIe x8 25G dual port SFP28 HHHL	2
TPM	1HY9ZZZ063G	Nuvoton SPI 2.0 FW1.3.1.0	1
IBBU	1HYQZZZ0031	QuantaGrid D52B/QuantaGrid D52BQ QS Super Cap Kit for QS 3516 SAS Mezz	1

5.3: Physical Network Setup

When setting up an OCN production environment on-premise, we recommend the following configuration:

- Clusters that host multiple pods should use a leaf-spine topology when interconnecting pods or physical servers.
- At least one leaf switch should be provided for each pod, having interface speeds matching server requirements listed in the previous section.
- Leaf switches must provide interfaces matching the physical server requirements (listed in the previous section) and northbound (spine connections) of 100 Gbps connections.
- Spine switches must provide the corresponding 100 Gbps interfaces to each leaf switch. The minimum requirement is one spine switch.

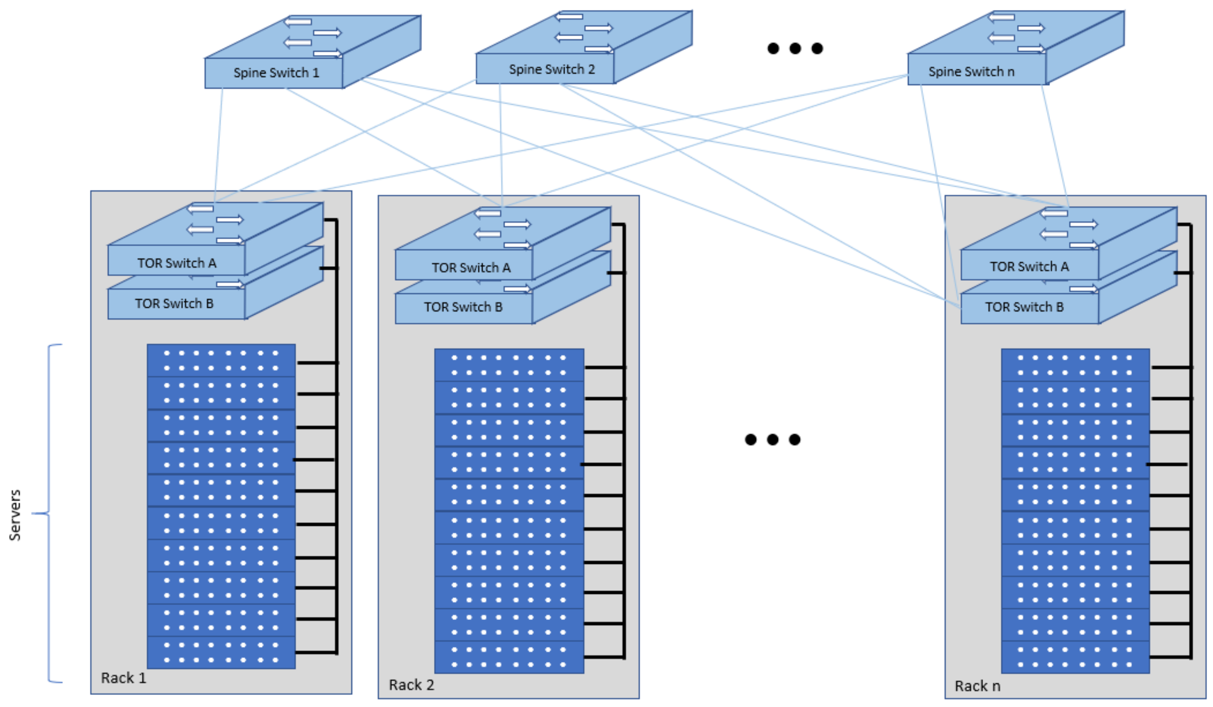

Figure 5 – Physical network setup

- A TIP Labs' staging environment setup for OCN in (as well as for operator pilots) can use only a single top-of-rack switch since hardware HA isn't a concern.
- The minimum networking configuration must provide at least five VLANs to partition the various networks
- The pod network topology should provide at least two networks with pre-allocated IP addressing schemes for out-of-band management network and the public network. The public network must be able to reach or access the public internet.

Network Name	VLAN ID	IP Subnet
Out of Band Mgmt (IPMI/iDrac/ILO)	100	10.1.100.0/24
PXE/Management Network	101	10.1.101.0/24
Control Network	102	10.1.102.0/24
Data Network	103	10.1.103.0/24
Public Network	104	10.1.104.0/24

5.4: Kubernetes Cluster High Availability

To stay conformant with requirements defined by CNTT member operators, OCN K8s architecture must support high availability. These requirements are as follows:

- Each availability zone or fault domain must run at least one master and one worker node to ensure HA of Kubernetes control plane services and resilience of workloads it manages.
- Each master node must run kube-apiserver, kube-scheduler, kube-controller-manager.
- Either three, five, or seven nodes must be running with the etcd service. They can be colocated on master nodes or run on separate nodes, but cannot run on worker nodes.

OCN K8s implements control plane services that are highly available and work in active-standby mode. All control components run on every K8s master node of a cluster, with one node at a time selected as a master replica and the others running in standby mode.

Every master node runs an instance of kube-scheduler and kube-controller-manager. Only one service of each kind is active at a time, while the others remain in the warm standby mode. The kube-controller-manager and kube-scheduler services natively elect their own leaders.

API servers work independently while an external or internal K8s load balancer dispatches requests between them. Each of the three master nodes runs its own kube-apiserver instance. All master node services work with the K8s API locally, while services running on the K8s nodes access the API by directly connecting to a kube-apiserver instance.

The following diagram illustrates the API flow in a HA K8s cluster. Each API instance on every master node interacts with each HAProxy instance, etcd cluster, and each kubelet instance on the K8s nodes.

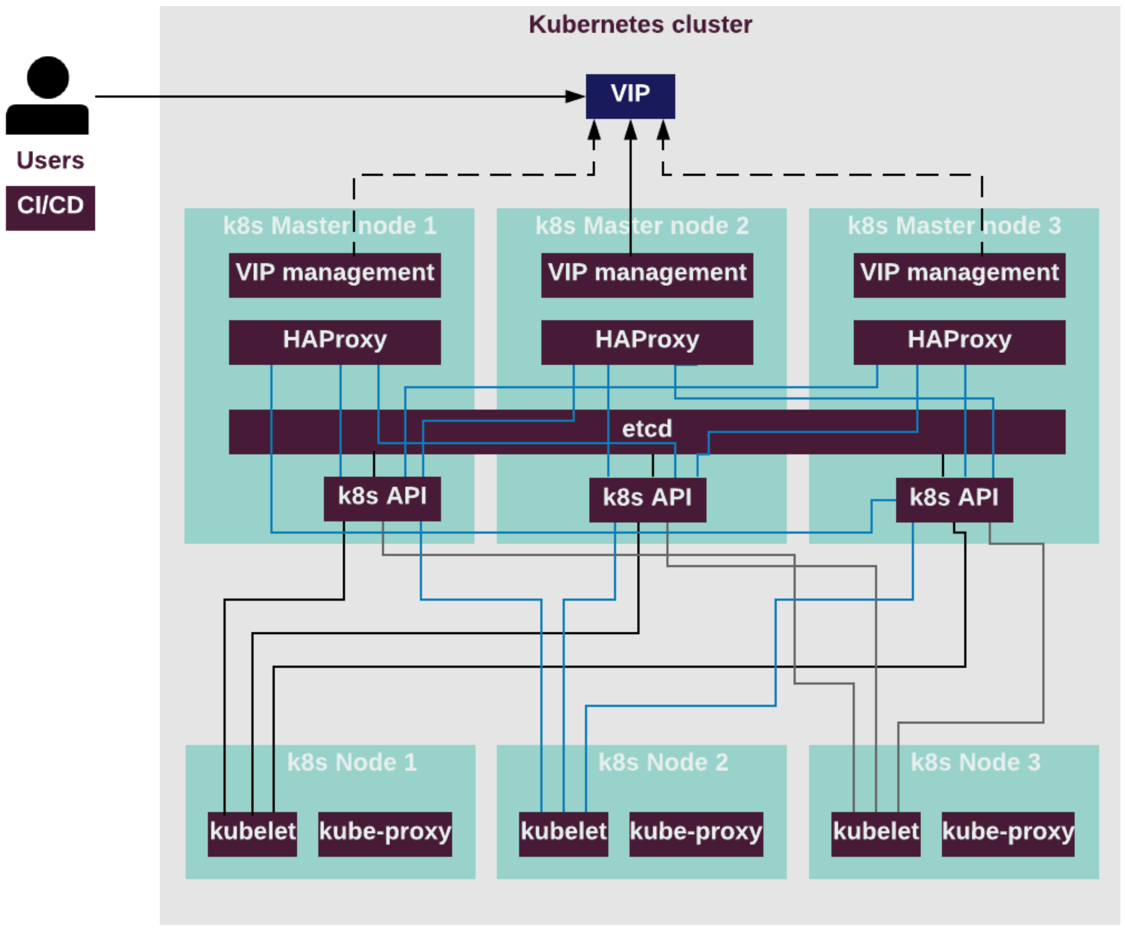

Figure 6 – API flow in a HA K8s cluster

Proxy server HA is ensured by HAProxy. It provides access to the K8s API endpoint by redirecting requests to kube-apiserver instances in a round-robin manner. The proxy server sends API traffic to available backends, while HAProxy prevents the traffic from going to unavailable nodes. The keepalived daemon provides VIP management for the proxy server. Optionally, SSL termination can be configured on HAProxy, so traffic to kube-apiserver instances goes over the internal K8s network.

Chapter 6: Public Cloud Deployment Considerations

Some operators might choose to deploy OCN in either a public or hybrid cloud environment. Since all public clouds are somewhat different in terms of hardware, network, and a catalog of infrastructure services they provide, it's beyond the scope of this document to prove a detailed deployment specification for each public cloud vendor. However, by using K8s and containers as core infrastructure building blocks, coupled with the guidelines in this chapter, we aim to provide a great degree of OCN portability across public clouds.

To optimize for cloud-to-cloud and cloud-to-bare metal portability, it's important to avoid the use of higher level, PaaS-like services provided by any given cloud provider (e.g., managed databases, cloud-managed K8s implementations, managed load balancers).

For example, if you're deploying OCN on AWS, you should limit use of its services to EC2 virtual machines (VMs), or bare metal servers with CNTT-compliant Kubernetes deployed on those VMs—rather than using a high-level service such as Amazon Kubernetes Service along with other AWS-managed services.

OCN microservices might require databases and other stateful components to run that don't come out of the box with CNTT-conformant, K8s-reference architecture. In this section we describe the OCN infrastructure approach to address these issues.

6.1: Managing OCN Databases in the Cloud

OCN relies on a number of stateful components (such as Redis and MySQL databases) to save configuration parameters and data. Magma Orchestrator is one of the seed OCN projects that uses PostgreSQL, MySQL, and Redis; it's a good example of such a requirement. When you build or run OCN components, we recommend minimizing database diversity so as to optimize for portability and reduce operational overhead. To that effect, we recommend using MySQL exclusively as a relational database and Redis as a distributed database for OCN service implementation and future operation.

To avoid using a cloud-specific implementation of the above databases, the OCN automation workstream implements and maintains up-to-date images and Helm charts for deploying them in a highly available configuration as K8s services. Helm charts and images of the highly available MySQL database will be maintained by OCN as part of the OpenStack-Helm project in this GitHub repository.

6.2: Persistent Volumes and Storage in the Cloud

Various OCN components, including the database services described above, likely require the use of persistent volumes. To guarantee persistence, all K8s deployments used for running OCN microservices must use K8s hostPath PersistentVolumes in combination with PersistentVolumeClaims and StatefulSets. See K8s documentation for information on configuring K8S pods to use persistent volumes.

Extending beyond K8s configuration, OCN components likely require a persistent storage backend (e.g., a dynamic volume manager such as Amazon EBS or object storage such as S3). (Deploying and managing an open source, distributed storage system such as Ceph in the cloud makes little sense.)

Moreover, when it comes to deploying in operator datacenters, operators might choose to use different vendor solutions as a distributed storage backend. Despite the detrimental effects to portability, OCN doesn't require the use of any particular storage backend, and explicitly permits use of persistent storage cloud services. Be aware that deploying to a public cloud and using a persistent storage service can result in decreased portability, as well as introduce potential data gravity issues to the extent that storage services are used.

6.3: K8s Networking in the Cloud

A main challenge of using cloud-specific K8s implementations is the varying approach to K8s networking among cloud providers. To that effect, OCN doesn't rely on cloud-managed K8s and implements a cloud-agnostic approach to K8s networking. For communication between services within a single K8s cluster, OCN implements ClusterIP service. For external access, we implement Traefic as the ingress controller. To ensure multiple network interfaces can access a single pod, OCN implements DANM as a CNI multiplexer.

In addition to the bare metal infrastructure staging environment at TIP labs, OCN maintains a cloud-agnostic K8s implementation hosted in AWS as an example that can be used by OCN microservice developers.

Chapter 7: Running OCN Components at the Edge

Some OCN use cases, such as fixed wireless access, might require a distributed deployment topology—where control plane components are centrally hosted with K8S underlay in the public cloud or in the operator datacenter—while data plane components such as UPF (or AGW coming from Magma seed code) are distributed across cell sites. For such use cases it might be impractical or cost-prohibitive to deploy a cluster of six servers at each cell site.

Running K8s at the far edge is an architectural decision that comes with pros and cons. Reasons to consider running it on edge devices include:

- Having a "container" as a single, universal unit for dependency encapsulation simplifies application lifecycle management. A single set of tools and workflows can apply system updates to components running at the core datacenter, as well as at edge devices.
- Having all user plane components containerized and managed by K8s makes it possible to automatically update individual application containers on a different lifecycle from the host operating system. This makes it possible to update components more frequently and with no downtime—even where only a single server or appliance is running at the edge.

Some of the reasons *not* to run K8s at the edge (particularly where edge is a single server appliance, such as the fixed wireless access use case) include:

- The primary benefit of using K8s is that it inherently maintains HA and seamless failover of all application components, protecting them from isolated hardware failures. This benefit largely disappears when running on a single appliance. If an edge appliance dies, all K8s services die with it.
- OCN use cases, such as fixed wireless, might require that an edge appliance to be extremely inexpensive and therefore limited in memory and compute resources. In some instances, OCN user plane components such as UPF or AGW might be embedded into the small cell itself. Therefore, dedicating some of these scarce resources toward the K8s control plane and away from data packet processing might not be advisable.
- Running dozens or hundreds of geo-distributed K8s clusters might make it easier to update individual appliance containers, but introduces the problem of having to update the K8s control plane services across the many locations.
- The closer you get to the network edge, the more diversity you see in relation to hardware platforms. For example, x86, arm64, and MIPs are all commonly used in edge devices. Running K8s across such a diversity of platforms is challenging.

- While there are a number of lightweight K8s implementations designed for the edge (e.g., kube-edge, k3s, and StarlingX), neither CNTT nor any of the other standards bodies have produced a common approach to running it at the edge.

In light of the above analysis and until further standardization happens among “far edge” K8s architectures, the OCN automation workstream recommends the following:

- In the public cloud or any site where five or more servers can be deployed, OCN must run on hardware using a cloud agnostic, CNTT-conformant K8s distribution as described in Chapter 5.
- For far edge sites where OCN components are to run on a single appliance, K8s shouldn’t be used.
    - Such sites should be updated by using immutable images
    - State and configuration should be persisted in the orchestrator
    - HA should be achieved through hardware—for example, if HA is required, another hardware appliance should be added at the cell site location