VIRTUAL EXPERIENCE
OCTOBER 11-14

2021 Fall
Technical Forum
SCTE® • NCTA • CABLELABS®

Implementing Multi-layer Infrastructure Management for Multi-Access Edge Computing (MEC) Services Using Kubernetes

A Technical Paper prepared for SCTE by

David K. Bainbridge
Senior Director, Software Engineering
Ciena Corporation
7035 Ridge Road
Hanover MD, 21076
dbainbri@ciena.com

Stephane Barbarie
Software Engineer
Ciena Corporation
7035 Ridge Road
Hanover MD, 21076
sbarbari@ciena.com

Dmitri Fedorov
Embedded Software Engineer
Ciena Corporation
7035 Ridge Road
Hanover MD, 21076
dfedorov@ciena.com

Marco Naveda
Senior Director, Network Architecture
Ciena Corporation
7035 Ridge Road
Hanover MD, 21076
mnaveda@ciena.com

Raghu Ranganathan
Principal & Distinguished Engineer, Advanced Architecture
Ciena Corporation
7035 Ridge Road
Hanover MD, 21076
rraghu@ciena.com

Table of Contents

Title **Page Number**

List of Figures

Title **Page Number**

1 Introduction

Communications networks are at the heart of advancing society and bringing people and places closer together. The evolution of communications services will be central in transforming how we work, play, collaborate, and interact with the environment around us. Emerging collaboration technologies such as augmented and mixed reality (AR/MR) promise to offer highly immersive, multi-user, real-time and content rich experiences that will simplify business operations, improve productivity, and unlock new services and revenue sources across a wide range of verticals. This type of application relies on large amounts of bandwidth and extremely low network delay to do real-time processing of very large data sets and tracking user and virtual object movement, while enabling fine-grained interactions between remote users, the physical world, and holographic objects. This will be possible as network application intelligence and cloud platforms converge at the network edge in Multi-Access Edge Computing (MEC) locations.

Over the last decade, communication service providers (CSP) have invested in significant network modernization to keep up with a growing demand for bandwidth hungry applications and increasingly distributed service consumption patterns. The adoption of Telco Cloud architectures for virtualizing network services has improved the operational responsiveness of the network. However, despite advances in network automation, the traditional top-down BSS/OSS operating model has not adapted to the realities of delivering dynamic, cloud-native network services to meet the needs of distributed MEC applications. This new application delivery paradigm requires new operational tools that enable CSPs to maintain carrier-grade operations for virtual machine-based virtual network functions (VNF), while evolving to the on-demand and intent-based deployment of cloud-native containerized workloads for the next generation of network services and MEC applications.

MEC infrastructure and connectivity services are expected to be a growing revenue source for service providers who build a distributed edge compute network platform for application delivery from cloud to edge to the customer premise [11]. However, no single provider will be able to address this massive opportunity, thus, there will be a need to coordinate resources across multiple layers of the network infrastructure as well as the federation of services from different providers in the wide area network (WAN). This type of inter-provider coordination requires the flexibility to define a specific network topology for a given application and user endpoints as well as exposing Telco Edge Operator Platform capabilities [9]. Such a system must include a tighter coupling with application networking for optimal placement of MEC workloads given the specialized requirements for compute resources and proximity to endpoints.

As enterprises adopt hybrid, multi-cloud strategies to support their digital transformation initiatives, pressure is mounting on the traditional telco cloud environment to align with the same level of service agility and developer experience offered by the hyper-scaler cloud providers. This is driving the need to support multiple providers for different components of the application infrastructure. For example, a cloud provider could be responsible for portions of the application infrastructure while a network provider could be responsible for portions of the network infrastructure and latency-constrained application components. Additionally, the use of the same MEC environment to support components from multiple application providers will require different hard or soft isolation techniques at MEC locations. A service provider must also find ways to align to a cloud provider’s edge deployment and operations model with suitable hooks for tighter visibility and control of the service provider’s network. Service providers will need tools to do this at scale with the likely need to manage 100s-1000s of highly distributed sites.

In this paper, we propose a Kubernetes-based control plane with built-in intent-driven automation to address these operational challenges and facilitate deployment of both virtual machine-based and container-based network functions (VNFs/CNFs) alongside MEC applications in a hybrid, multi-cloud architecture with a multi-layer connectivity network underlay.

This paper first describes the use of Kubernetes technologies and extensions to those technologies introduced as a solution to address the operational challenges implementing a MEC architecture (section 2). These technologies are then applied as the paper describes deploying a MEC architecture based on the Kubernetes system (section 3). Finally, the findings and recommendations based on the work completed are summarized (section 4).

2 Background and Technologies

The European Telecommunications Standards Institute (ETSI) [1] provides a reference architecture for the deployment of MEC hosts and applications. With the shift to containerized workloads in a cloud-native environment, ETSI [7, 8] work supports the use of container management systems such as Kubernetes for providing platform-as-a-service (PaaS) services. Additionally, the mapping of the ETSI management and orchestration (MANO) information model to the container workload deployment model enables an approach for implementation using a Kubernetes model. The MEC architecture can be deployed using a Kubernetes model and, with extensions, a more complete MEC model can be achieved with virtual machines (VM), VM to container service chaining, and constraint policy-based connectivity.

ETSI [Section 8 of reference 1] defines MEC service as a service provided and consumed either by the MEC platform or a MEC application. In the context of this paper, the term is inter-changeably used for both user application services, e.g., a MEC application like AR/VR rendering, as well as host or platform services, e.g., traffic management service or domain network service. As an example, some host level MEC services could be offered by a network provider while some application level MEC services could be offered by a cloud provider. Some of these MEC services may be part of the Kubernetes implementation.

2.1 Kubernetes In Brief

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates declarative configuration and automation. It has a large, rapidly growing ecosystem. It provides a framework to run distributed systems resiliently, providing standard patterns for application deployment, scaling, failover, security, and load balancing. A full description of Kubernetes and its capabilities can be found at [10].

Since Kubernetes primarily operates at the container level rather than at the hardware level, it provides some general features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. There are also extensions to Kubernetes that allow it to manage VM based workloads which will be discussed later in this section of the document.

Kubernetes is not monolithic, and its default solutions are optional and pluggable. It provides the building blocks for building platforms but preserves user choice and flexibility where it is important.

A single Kubernetes deployment is known as a cluster and consists of a set of machines (physical or virtual) called nodes, which are utilized to host containerized applications. The nodes within a cluster can be classified as either a control-plane node, where workloads that implement the Kubernetes system are executed, or a worker node, where primarily application workloads deployed to the cluster are executed.

The Kubernetes control plane manages the worker nodes and the pods in the cluster, and it makes global decisions about the use of cluster resources. A Kubernetes pod is a schedulable entity that is comprised of one or more containers. The control plane components can be run on any machine(s) in the cluster, however, the usual practice is to run all control plane components on one or more machines and avoid running user containers on the same machines as the control plane components.

The following illustrates Kubernetes cluster elements described above [12].

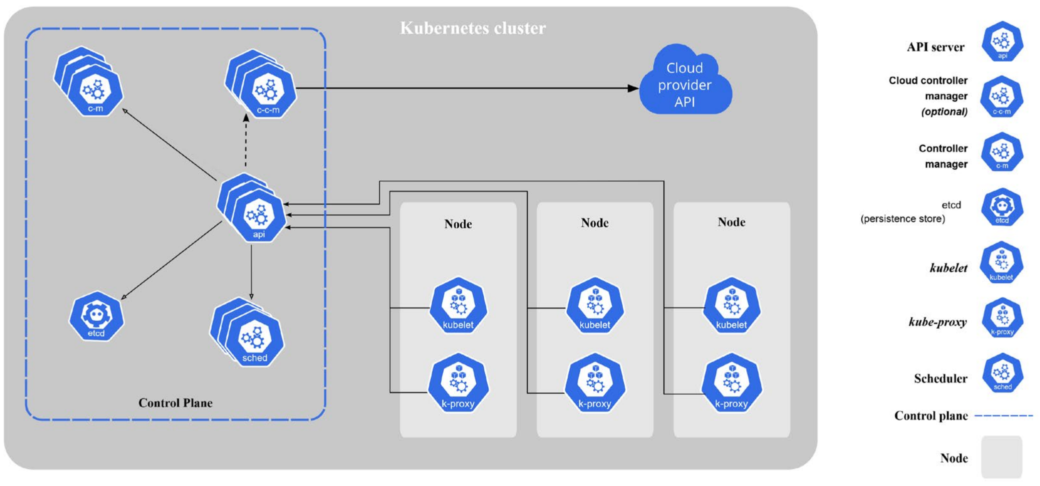


Figure 1 - Kubernetes cluster with all its components

As mentioned, Kubernetes is an extensible system and a key mechanism to extending Kubernetes is implementing a custom resource definition (CRD). A resource created through this feature can be used to store and manipulate information in the Kubernetes system. These custom resources are normally used in combination with a custom Kubernetes controller that interprets the data for the custom resource type contained in the Kubernetes store and then reacts to changes in the data (adds, deletes, modifications). This extensibility via CRDs highlights the fact that at its core, Kubernetes is a declarative based resource management system, and this can be leveraged when implementing a MEC architecture with Kubernetes.

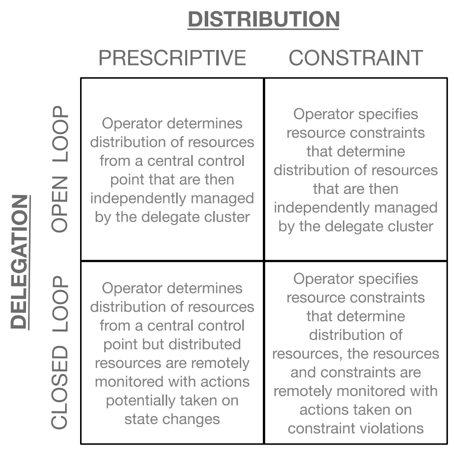
2.2 Multi-cluster Strategies

Enterprises are adopting Kubernetes as a platform to enable application portability and agile deployment across public clouds, private environments, and more importantly on the network edge to optimize local service performance. This is critical for enterprises running retail, hospitality, and manufacturing operations with 100's if not 1000's of locations where application infrastructure is needed to support business to consumer (B2C) and business to business (B2B) applications, as discussed in section 2.8.

Kubernetes supports mechanisms such as pods and name spaces to isolate application components, and ensure resources are allocated optimally within a multi-tenant edge cluster. However, as Enterprise MEC applications proliferate at the network edge, industry trends are starting to emerge to define mechanisms to spread workloads across multiple clusters in different geographic areas. The chief technical reasons for multi-cluster deployments are:

- Lower latency by deploying applications closer to end users
- Service availability with fail-over support and geo-redundancy
- Workload scalability across distinct physical clusters with specialized resources
- Workload isolation & security with physical separation

The main two dimensions of these multi-cluster trends are the *distribution* of an application's resources and the *delegation* of lifecycle control of the distributed application resources, (see Figure 2).



DELEGATION	DISTRIBUTION: PRESCRIPTIVE	DISTRIBUTION: CONSTRAINT
OPEN LOOP	Operator determines distribution of resources from a central control point that are then independently managed by the delegate cluster	Operator specifies resource constraints that determine distribution of resources that are then independently managed by the delegate cluster
CLOSED LOOP	Operator determines distribution of resources from a central control point but distributed resources are remotely monitored with actions potentially taken on state changes	Operator specifies resource constraints that determine distribution of resources, the resources and constraints are remotely monitored with actions taken on constraint violations

Figure 2 - Multi-Cluster Scheduling Strategies

Distribution of an application's resources refers to how an operator specifies the initial distribution of the resources across the available clusters. Distribution may also reference how an application's resources are redistributed based on a failure or other event. In a *prescriptive* system, the operator specifies the cardinality and location (Kubernetes cluster) for each application resource. In a *constraint-based* system, the operator specifies the constraints for application resources, such as CPU, memory, network bandwidth, and network latency to an internal application resource or another external MEC application. These constraints are used by a scheduler to determine the optimum placement of the application resources.

Delegation of application resource lifecycle control refers to how the lifecycle of a resource is managed, including initial assignment to a member cluster, and any reassignment to a different cluster based on manual intervention or an event. In an *open-loop* system, once a resource is delegated to a participating cluster, the resource's lifecycle is completely managed by that cluster and will never be removed from that cluster except via an explicit action. In a *closed-loop* system, once the resource is delegated to a participating cluster, a feedback loop is used to monitor the resource and decisions about moving a resource would be based on a defined policy.

Both the distribution and delegation dimensions reflect the level of automation in a multi-cluster system. Systems that fall to the lower right quadrant (see Figure 2) tend to be more autonomous where systems that fall to the upper left quadrant tend to be configuration systems that strictly enact actions in the exact way the operator specifies without any remediation based on failures or resources violations.

2.3 Node/Cluster Capability Discovery

Applications are increasingly looking to leverage available hardware accelerators (GPUs, TPUs, etc.) and software data plane technologies (DPDK, VPP, etc.) to meet their performance requirements. This information is useful to a MEC control plane when placing MEC service components on inter-connected compute nodes. Leveraging Kubernetes and CNCF projects enables the deployment to self-discover a node's capabilities and report or expose those capabilities to a control-plane to be used during scheduling of workloads. Specifically, the CNCF node feature discovery project [55] provides this capability by discovering node features and labeling nodes in a standard format to allow features to be used as part of the standard Kubernetes scheduling capability. This is of critical importance in space & power constrained MEC environments, where full visibility of resource capabilities and programmability of the network infrastructure enable optimal allocation of premium resources.

2.4 Constraint Policy

Kubernetes provides a mechanism for a workload to specify resource constraints that can be used by the control-plane to influence the node selected when scheduling a workload. The existing mechanism is simplistic and predefines only the CPU and memory resource. Kubernetes does allow for other resources to be defined but limits the requested value of those resources to be an integer without a unit specification.

The system we have developed defines a constraint policy model that allows for arbitrary resource constraints to be specified and then leveraged by a custom scheduler as well as de-scheduler [6] extension.

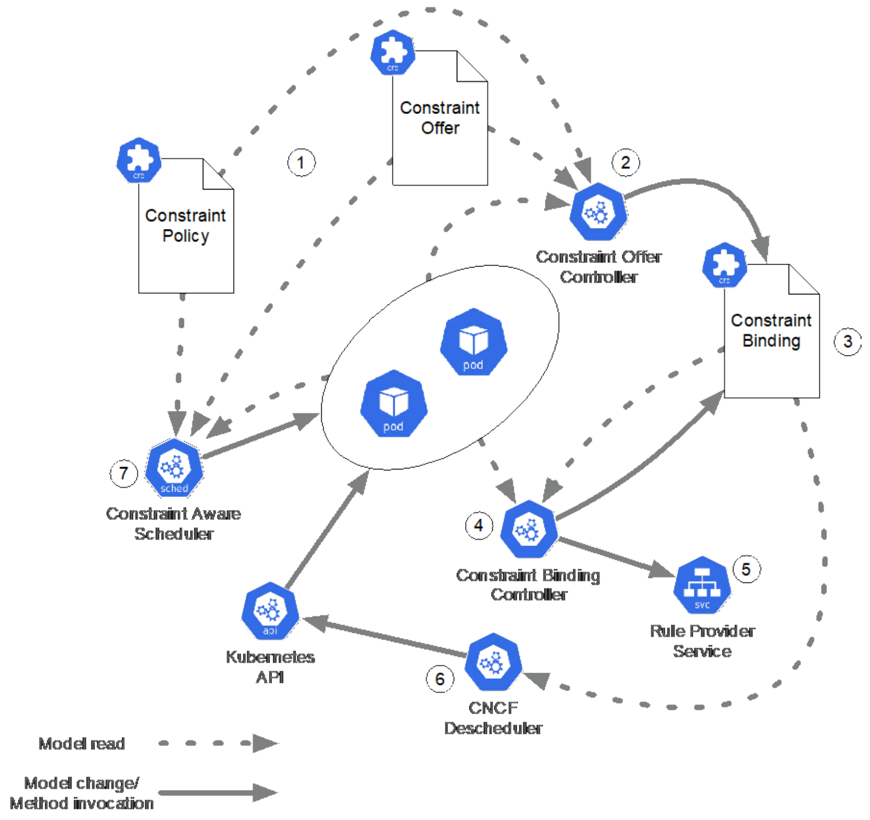


Figure 3 - Constraint Policy Overview

Figure 3 depicts the model and interaction of the constraint policy extension. A constraint policy is a set of constraint rules, where each rule is a tuple of constraint name, constraint request, and constraint limit. The constraint subsystem is designed to be dynamically extensible, and as such the constraint name is a moniker that is used to locate a constraint provider implementation at runtime. The implication is that the system does not pre-define any set of constraints. The constraint request and limit mirror the semantics of the existing resource constraints in Kubernetes in that a request is the preferred value, and the limit is the "worst" allowed value.

A Constraint offer ① is used to associate a constraint policy to one or more workloads (Pods, Services, or NSM network chains in the current implementation). The association is discovered by the Constraint Offer Controller ②, using a selector based on the tuple of Kubernetes ApiVersion, Kind, and Name. For each association discovered, a Constraint binding ③ is created to track the specific policy-workload association including its compliance status. Offers are periodically evaluated and the set of bindings is updated accordingly, deleting bindings that are no longer valid and creating those that now exist.

The Constraint Binding Controller ④ periodically evaluates the policies against the list of bindings leveraging the various provider services ⑤. A provider service is identified via a well-known label based on the constraint name. This allows providers, and thus constraint types, to be dynamically managed. The compliance of a binding is updated as part of the status value for the binding.

This implementation extends the CNCF de-scheduler project ⑥ to monitor the status of the bindings and when a binding is found to be non-compliant, the de-scheduler may, based on a policy setting, evict the pod via the Kubernetes API ⑦.

2.4.1 Network Connectivity Constraints

Utilizing the constraint policy capability, this implementation defines a set of network connectivity-based constraint providers: bandwidth, latency, and jitter. Using these constraints, an operator can specify the requirements for connectivity between two or more workloads.

By implementing a declarative model for connectivity within Kubernetes, the network becomes part of the overall resource model within the environment, opening new automation use cases, including the ability to declaratively specify constraints that affect the underlay and overlay networks to meet the operator specified application requirements.

2.5 Scheduling Optimization

By default, the scheduling context for Kubernetes is a single pod. During scheduling, Kubernetes selects a node and assigns the pod to that node. Once a pod is assigned to a node, the containers defined within the pod are created and invoked. This can lead to sub-optimal scheduling when constraint policies (see Section 2.4.1) represent a binding between two or more pods as is the case with a connectivity constraint. For optimum scheduling, the entire set of connected pods to be scheduled should be known, and a “plan” should be created such that the pods can be scheduled according to the optimized plan.

To provide this capability, a scheduler extension was developed that operates as an optimized schedule plan builder, as well as a gating function to prevent pods from being scheduled until a trigger is detected. For the initial implementation, we are using a “quiet” timer, but this is easily extendable to support additional trigger types. The quiet timer simply fires when no new pods are defined over a specified period, thus the assumption being that all required pods have been defined and an optimal schedule can be produced.

Before the trigger fires, the scheduler is called repeatedly for each pod. When the planner is invoked, it queries the list of all unscheduled pods and creates a candidate plan, utilizing any specified constraints via the constraint policy resources. If the candidate plan is preferred over the existing plan, the existing plan is replaced by the candidate plan. In either case, an empty node list is returned indicating to Kubernetes that the pod cannot be placed at this time and the pod will remain in a “Pending” (non-assigned) state. After the trigger fires and the scheduler is called, the scheduler uses the plan to determine the node to which to assign the pod. As pods are assigned to nodes, they will be instantiated, and their containers will be created. At this time the connectivity-based constraint policy bindings will be created based on the scheduled pods.

2.6 Network Controller

As described in previous section, the scheduler extension developed as part of this work can leverage the connectivity-based constraints when scheduling workloads.

During development of the solution, while simulating network latency between two nodes, it was noticed that the connectivity-based constraints could not be met by the existing network configuration, causing workloads never to be scheduled, even though the underlying network had the capacity to meet those constraints. As these conditions could exist in a production deployment, especially a multi-cluster deployment, the concept of a network controller was introduced into the solution.

A network controller was represented by a defined interface and could be used by the scheduler to request network resources when the current network configuration could not meet the specified constraints. From the perspective of the scheduler, the network controller is an external entity located by a label on the Kubernetes service resource.

When invoked, the network controller has the flexibility to modify the underlay, overlay network, and/or network slicing to meet the resource request, returning to the scheduler enough information so that the pods that will be created can leverage the modified resources. This information can be used with the network service mesh project to ensure the connectivity to containers by creating the proper network interfaces and configuration on the containers. When the existing network does not meet the constraints and the network controller is not able to modify the network to meet the constraints, the pods will remain in the pending state until the constraints are modified, or the network comes into compliance.

The network controller was then integrated into the de-scheduler capability. When a connectivity-based constraint was found to be out of compliance, rather than immediately evicting the pod for rescheduling, a capability was added to the network controller to mediate this situation via network configuration. If the network controller is not able to bring the network back into compliance, then based on policy, pods may be evicted to be rescheduled or the violation can be ignored to prevent service disruption.

2.7 Common Application Function Model

Telco cloud implementations based on the ETSI network function virtualization (NFV) model have been in production for several years, delivering data and control plane network functions in a much more flexible and software-based format. These virtual machine-based VNFs evolved from the software applications delivered via dedicated hardware appliances for traditional switching, routing, firewall, and signaling services, among others. From a compute environment perspective, these network applications are no different than enterprise or consumer type applications that are delivered from cloud-native environments today, except for requiring specialized hardware assist for packet processing functions. These functions include protocol encapsulation and decapsulation, packet header classification, inspection and manipulation, wire-speed forwarding, encryption, and traffic protection, to name a few. These functions typically require traffic to be steered through multiple functional blocks that make up a network application service chain. The implication is that it is possible to describe a common model for service function chaining of network application components independently of the specific software logic running within these components. This service function chaining specifies the communication patterns and processing policy for function chaining blocks.

2.7.1 Service Function Chaining

A network function is composed of one or more deployable components. These components can be inter-connected workloads to provide the overall features intended by the network service. The workloads form a chain with traffic flow sequence dependencies, and therefore, should be deployed as a unit on a single compute cluster node for optimal performance. In a hybrid virtualization configuration, a network function could potentially inter-connect hypervisor and container-based workloads on the same cluster node. For this reason, the MEC compliant platform should expose a common network function model to onboard and chain different workload formats.

Although sub-optimal, there may be cases where service function chains span multiple nodes within the same cluster or even multiple clusters separated by an edge network. This may result from constraints imposed on the service chain and the availability of specialized resources such as GPU or smart NICs required to support hardware accelerated functions. In such cases, constraints such as network latency, bandwidth, packet delivery guarantees, and traffic balancing must be taken into consideration when composing the end-to-end service chain through cluster federation mechanisms and network connectivity constraint policy.

2.7.2 Kubernetes Controllers for Function Chaining

When a purpose-built device and associated objects that consume compute resources are not directly modeled by Kubernetes, they can be represented through CRDs and the Kubernetes API can be extended to expose the configuration and capabilities of that new device. A custom controller can then be implemented to manage the lifecycle and translate the resource data into instructions that the target device may understand. Once CRDs and controllers are installed in a Kubernetes cluster, the orchestration of the device can be done through the Kubernetes control plane by abstracting the interactions through the custom device controller.

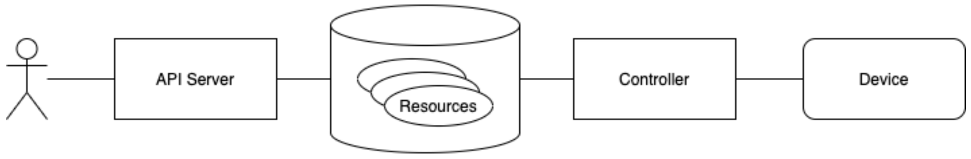


Figure 4 - Orchestrating Device Configuration via Kubernetes CRDs

We used this approach to support the orchestration of VM-based and container-based network functions (NF) on a common operational platform and service chained on a common network layer. This common orchestration framework was achieved by defining models to abstract the network function and chaining complexity. The models are then converted to Kubernetes custom resources with their corresponding controllers. The NF controllers are responsible for orchestrating a VM or container, based on the specified NF type. The chaining controller can instruct the cluster to establish connectivity between the NFs on the underlying data plane, which may consist of software-based switching and hardware-based traffic processing.

This orchestration walkthrough for VMs and containers is a simplified and high-level view of what really needs to occur. Two scenarios can be considered to configure a system that supports virtualization for both VMs and containers.

1. System with hypervisor engine only with a VM instance running container native constructs, such as Kubernetes, to host containers.
2. System with both hypervisor and container engines to host VMs.

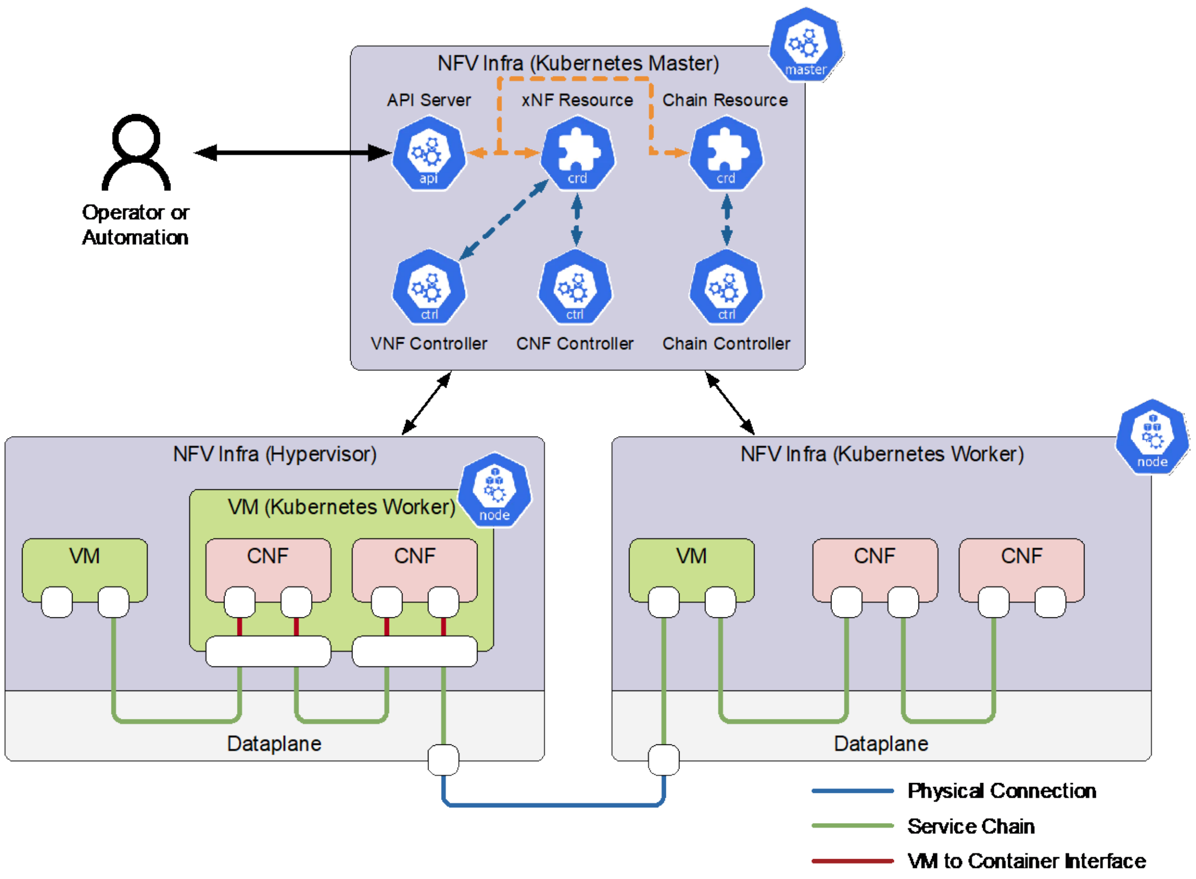


Figure 5 – CRD Driven Chaining

In the first scenario (Figure 5, left side), VMs are orchestrated through the system's hypervisor. A VM running Kubernetes is also used to deploy the container-based network functions. The connectivity between network functions is handled by service chaining the interfaces allocated to the VMs and then linking the containers with the Kubernetes VM interfaces using container networking interface (CNI) plug-ins (for example, Multus). In the second scenario, VMs and containers are orchestrated by their respective virtualization engines. The connectivity between the network functions is handled by service chaining the interfaces allocated to the VM and containers. In both cases, the interface allocation is provided by the data plane embedded in the NFV infrastructure.

In the second scenario (Figure 5, right side), the VM/Kubernetes capable infrastructure is used to create both the VMs and the containers. The connectivity between the network functions handled by using the infrastructure interfaces to chain the VM interfaces across the data-plane and then into the container-based NFs.

2.8 Public/Private Cloud Integration

When architecting a multi-cluster Kubernetes deployment, it is important to understand common industry deployment models. Currently, it is common that network operators and enterprises leverage both their private cloud resources and resources available from cloud hyper-scalers such as Google, Amazon, and Microsoft. Consider a scenario where an Enterprise customer with a large chain of retail stores is planning the introduction of new cloud native applications for inventory management, store security, advertising, and in-store customer engagement. This customer uses one of the major cloud providers to run their own DevOps environment and a national network operator to inter-connect all their stores to MEC locations, private data centers and cloud. A key requirement for the Enterprise IT operations team is to unify the management and delivery of containerized applications to 100's of locations (premise and edge) while maintaining a common network and security policy nationally. As Figure 6 illustrates, this leads to designing a fabric of Kubernetes clusters deployed at many locations, managed through a cloud provider's control-plane and interconnected by a network operator that hosts some of the clusters within the MEC locations.

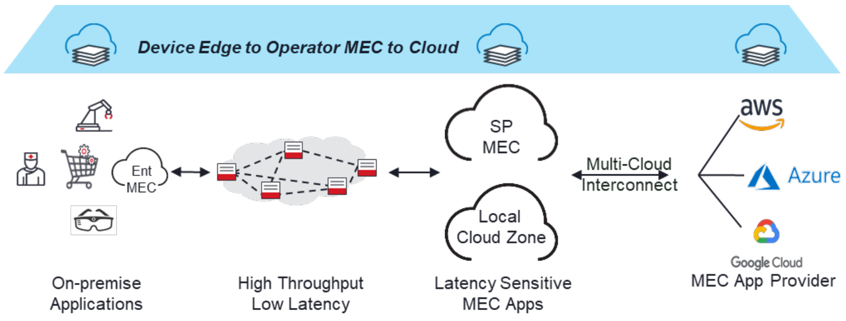
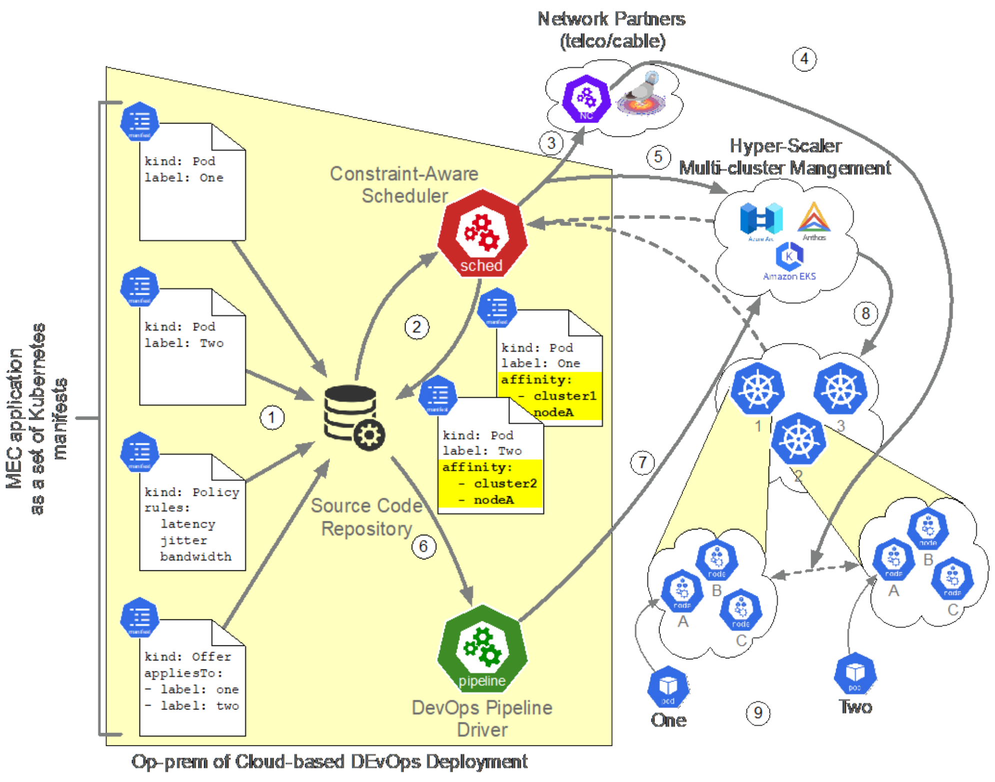


Figure 6 - Enterprise Multi-MEC Applications

While the capabilities described in the sections above can be deployed into Kubernetes clusters under the administrative control of the network operator, it is not always possible to deploy these capabilities on the Kubernetes clusters provided by the hyper-scalers. This is more obvious when it comes to the aggregation or federated level as each hyper-scaler typically provides a custom federation solution as one of multiple tightly integrated cloud-based services, i.e., Google Anthos, Amazon EKS, Microsoft Arc, Rancher, etc.

How these capabilities can be integrated with the various hyper-scalers' offering depends on the amount of customization each allows. In the case where a custom scheduler extension cannot be deployed, this can be "worked-around" by creatively assigning node affinity to resources before allowing the hyper-scalers scheduler to be activated. Node affinity is a standard Kubernetes capability available on all distributions.

This can be implemented by shifting the capabilities described above, particularly scheduling and de-scheduling, from the Kubernetes domain to a DevOps domain, as depicted in Figure 7. In this situation, the DevOps pipeline would be leveraged such that when an application is pushed to storage ①, the pipeline would evaluate the scheduling needs of the workload and augment the resources with the node assignment encoded as a node affinity configuration ②. Additionally, the constraint aware scheduler may, depending on availability, contact a network controller provided by the network provider ③ to request network capabilities compliant with the constraints specified in the constraint policy. This in turn might trigger the network provider to reconfigure the underlay network ④. If a network controller is provided by the hyper-scaler, then the scheduler may also make requests via that interface which could affect both the overlay and underlay ⑤. After the scheduler updates the manifests and commits those back to storage, the DevOps pipeline receives the augmented manifests ⑥ and pushes the manifests to the hyper-scaler managers ⑦. The hyper-scaler managers process the manifests using their standard schedulers ⑧, adhering to the standard affinity rules, and enact the set node assignment ⑨.


Network Partners (telco/cable)
NC
Hyper-Scaler Multi-cluster Mangement
Azure Arc
Anthos
Amazon EKS
Constraint-Aware Scheduler
sched
Source Code Repository
pipeline
DevOps Pipeline Driver
MEC application as a set of Kubernetes manifests
Op-prem of Cloud-based DEvOps Deployment
1
2
3
4
5
6
7
8
9
node
A
B
C
pod
One
Two

```
kind: Pod
label: One
```

```
kind: Pod
label: Two
```

```
kind: Policy
rules:
  latency
  jitter
  bandwidth
```

```
kind: Offer
appliesTo:
- label: one
- label: two
```

```
kind: Pod
label: One
affinity:
  - cluster1
  - nodeA
```

```
kind: Pod
label: Two
affinity:
  - cluster2
  - nodeA
```

Figure 7 - Using Constraint-Base Scheduling with Public Clouds

Other than the scheduler extension, the described technologies should be able to be leveraged within a hyper-scaler's environment as the other technologies either are common user-based extensions (CRDs + controllers) or components that run outside the core Kubernetes control-plane (de-scheduler). The one exception is the network controller, which may require support from the hyper-scalers to support underlay control, although it is possible to implement a network controller that only affects the overlay.

3 MEC Architecture on Kubernetes

In this section, we describe how the MEC architecture can be implemented using the de facto industry standard container orchestration system originally developed by Google, i.e., Kubernetes. Figure 8 depicts the standard MEC architecture on the left and on the right depicts that same architecture with an overlay that indicates the cloud native technologies that can be leveraged to implement the MEC architecture. The following sections will detail how each component of the MEC architecture can be implemented using specific cloud native technologies.

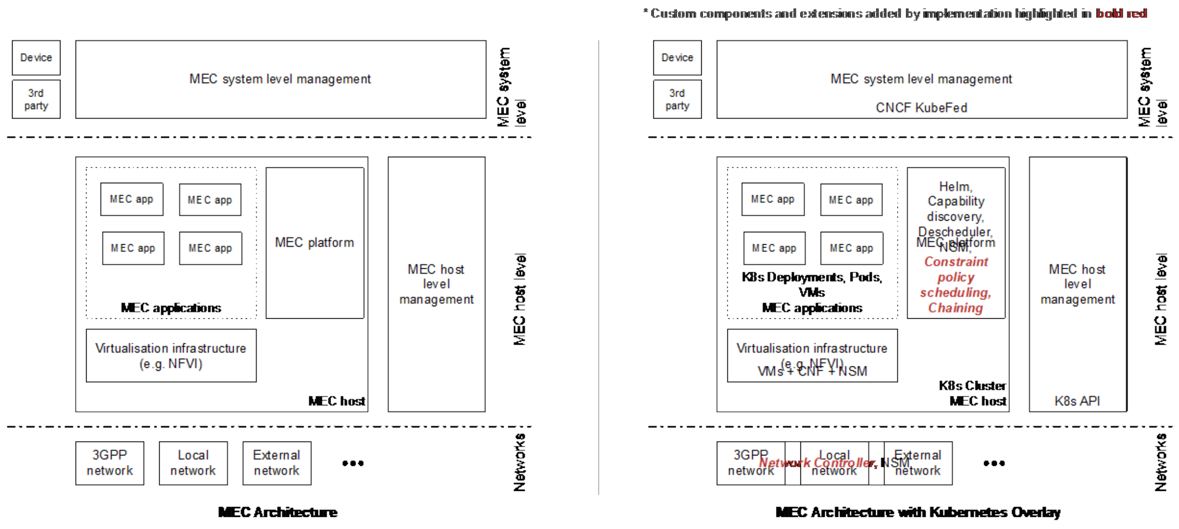

* Custom components and extensions added by implementation highlighted in bold red
Device
3rd party
MEC system level management
MEC system level
MEC app
MEC applications
MEC platform
Virtualisation infrastructure (e.g. NFVI)
MEC host
MEC host level management
MEC host level
3GPP network
Local network
External network
•••
Networks
MEC Architecture
CNCF KubeFed
K8s Deployments, Pods, VMs
Helm, Capability discovery, Descheduler, NSM,
Constraint policy scheduling, Chaining
VMs + CNF + NSM
K8s Cluster
K8s API
Network Controller, NSM
MEC Architecture with Kubernetes Overlay

Figure 8 - MEC Architecture with Kubernetes Overlay

3.1 MEC Host

A MEC host is defined as “an entity that contains the MEC platform and a Virtualisation infrastructure which provides compute, storage and network resources for the MEC applications. The Virtualisation infrastructure includes a data plane that executes the traffic rules received by the MEC platform and routes the traffic among applications, services, DNS server/proxy, 3GPP network, other access networks, local networks and external networks.” [1]. By this definition the MEC Host is functionally equivalent to a Kubernetes cluster, which is defined as “A set of worker machines, called nodes, that run containerized applications” [2] The Kubernetes cluster provides the virtualisation infrastructure and data plane as required by the MEC definition. While Kubernetes’ original focus was orchestration of containers, several virtualizations extentions have been added to Kubernetes to provide a run-time for virtual machines and networking.

3.1.1 MEC Virtualization Infrastructure

The virtualization infrastructure of a MEC compliant system should have the ability to orchestrate hybrid service deployments where VMs and containers can coexist on the same platform. To achieve this hybrid configuration, there is a need to accommodate VM based network functions within Kubernetes. A system with such capabilities would have to share resources (compute, memory, storage, networking) to offer a seamless integration with the hosting platform. KubeVirt is one of several projects of the Kubernetes ecosystem that can manage the lifecycle of virtual machines within a Kubernetes cluster while also supporting container workloads.

Additional solutions and platforms exist that provide VM/container capability through the Kubernetes declarative model solution. Further, some of these solutions provide a tight integration with the network interfaces such that they are purpose built to support VM and container-based network functions.

Today the declarative models used to define VM based resources vary across the available solutions and most focus primarily on the detailed attributes for creating a VM and less on the concept of chaining NFs. The solution proposed in section 2.7 bridges this gap by allowing the specification of network services that can contain both VMs and container-based NFs, abstracting away the specific virtualization choice and focusing on the connectivity between those functions. Further, this approach can be extended in the future to support additional virtualization techniques and/or new infrastructure as it is released by vendors.

3.1.2 MEC Applications

A MEC application “runs a virtualized application … on the infrastructure provided by the MEC host” [1]. Within Kubernetes, the typical executable workload is known as a pod, which is a set of containers run on a single Kubernetes node that share storage and networking. As shown above, with the use of CRDs, Kubernetes can be extended such that a workload may be either a pod (container) or a VM.

Kubernetes provides several “higher” level resources constructs that help the operator group and deploy the basic building blocks of an application. These include a Deployment, which is a set of distinct pod definitions and the cardinality for each of the pod types, as well as a ReplicaSet, which maintains a stable set of pod instances for a single pod definition. In addition to pods and other deployment constructs, Kubernetes also provides mechanism to enable load-balancing and high availability for applications.

These basic building blocks provided by Kubernetes provides the basis on which MEC compliant applications can be built. Because an application typically requires more than a single Kubernetes resource, a higher-level application construct can be created using the CNCF Helm [3] tool. This abstraction allows a MEC application developer to specify any number of Kubernetes resources as a set and then deploys that set of resources under a single name, thus allowing a complete application to be deployed instead of dealing with the applications piecemeal.

At its core, Helm is a template engine that create instances of resources based on defined templates, parameterized Kubernetes resource definitions, substituting configurable values for the required parameters. Templates can be core Kubernetes resources as well as CRD defined resources, thus providing access to the full resource model.

3.1.3 MEC Host level management

As described above, the MEC virtualization infrastructure capability can be provided by Kubernetes with extensions to support VMs. Through the defined NF CRDs, both VM and container-based capability can be specified and deploy via a common abstraction. Once deployed, the Kubernetes control-plane will monitor the lifecycle of the resources accounting for scalability and high availability. Additionally, Kubernetes provides a security and network infrastructure to support application deployment.

With the additional of the connectivity-based constraints, scheduler extensions, de-scheduler, and network controller, Kubernetes provides the base capabilities required of MEC host level management.

3.1.4 MEC Host Level Scheduling

MEC host level scheduling is the equivalent of scheduling on a single Kubernetes cluster. The previously described technologies (constraint-based scheduler, optimized scheduler, de-scheduler, and network controller) work in concert to provide the scheduling of workloads to nodes.

3.1.5 MEC Host Level Networking

While a single Kubernetes host provides basic MEC host networking capabilities, through the use of add-on capabilities such as the network service mesh (NSM), additional MEC host (or Kubernetes intra-cluster) networking can be leveraged. A key consideration when deploying a NSM into a Kubernetes cluster is the ability to declaratively define the network connectivity such that there is a separation of concerns between the development of the application and the deployment of the application., i.e., the expected connectivity should not be “baked” into the application code and instead be left to deployment (declarative) configuration. This can be achieved with the NSM implementation.

A network function deployed within a MEC host must focus on serving its intended purpose and should remain unaware of any chaining requirements with other network functions. The Network Service Mesh framework (NSM) in the Kubernetes eco-system fills the role of creating chains and managing the assignment of a network function within a chain. It does so by augmenting orchestrated network functions with a sidecar container and controlling the interactions between the sidecars. The NSM manager can then implement the desired topology by establishing links between sidecars through the NSM data plane.

3.2 MEC System Level Management

The Cloud Native Computing Foundation (CNCF) has defined a special interest group, the Multi-cluster Special Interest Group (SIG), whose charter specifies that this SIG focuses on "solving common challenges related to the management of multiple Kubernetes clusters". [4] As indicated above, if a MEC Host represents a single Kubernetes cluster, then the MEC system level management is meant to manage multiple MEC Hosts and thus multiple Kubernetes clusters.

The CNCF Multi-cluster SIG facilitates the development of a solution for deploying workloads across multiple Kubernetes clusters known as "KubeFed". KubeFed allows an operator to specify the cardinality and location of workloads that are part of an application. Thus, an operator can deploy an application and prescriptively control which cluster a pod is deployed on and how many instances of that Pod are deployed to that cluster. While this is required, it is not sufficient for an autonomous MEC system that can deploy MEC services based on their compute, storage, network, and other resource constraints. Sections 2.4, 2.5, and 2.6 describe a constraint policy extension to Kubernetes that can be applied to a multicluster Kubernetes deployment to provide the capability to deploy MEC services across multiple MEC hosts based on operator specified constraints.

3.2.1 Why Multi-MEC Host is needed

In modern deployment architectures, a single MEC host is not always sufficient to meet the MEC service requirements for latency and/or performance. MEC services will be designed around network and performance bottlenecks, but these designs cannot always compensate for the limitations imposed by the constraints of a single MEC host.

To truly meet the requirements of modern and near future MEC services, deployments must take advantage of multiple MEC host deployments where some of the MEC hosts may be "network close" to the end client with lessor compute power, commonly called edge, and other hosts may be "network distant" with greater computer power.

It is important when deploying a MEC application across multiple MEC hosts that the MEC application is not "topology aware" in that it is not aware of the network location of the compute nor the network on which it is deployed. Instead the MEC applicaiton must specify the constraints it requires and allow the "MEC control-plane" to allocate resources to meet the specified constraints. Providing this separation of concerns between the MEC application and the MEC control-plane allows operators to better align their resources and provide the expected quality of service (QOS) to their clients.

3.2.2 MEC and Edge Computing

Edge computing is the delivery of computing capabilities to the logical extremes of a network to improve the performance, operating cost and reliability of applications and services to the user of the services.

As ETSI GS MEC 003 document [1] states, "Multi-access Edge Computing enables the implementation of MEC applications as software-only entities that run on top of a Virtualization infrastructure, which is located in or close to the network edge."

While this means that MEC can define a network edge capability, it is also true that MEC can define a metro or central data center. In the context of MEC, any Kubernetes cluster is considered a MEC host regardless of the "nearness" to any given client. As such, at the system level, MEC hosts create a mesh of connectivity that can be leveraged by users that deploy MEC applications.

A MEC *location capability* is simply the MEC host that is "network near" the client of a given MEC application. Thus, any given MEC host may be *at the edge* to some client regardless of its actual location.

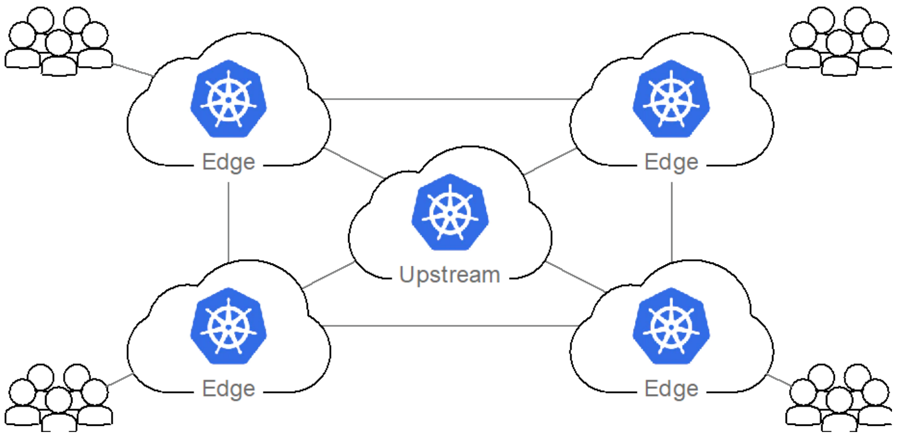


Figure 9 - Depiction of a Kubernetes Cluster Mesh

The illustration above shows an example of how Kubernetes clusters can be inter-connected to represent a MEC host mesh. While some of the clusters are labeled "Edge" or "Upstream" it is important to note that all clusters are functionally equivalent. Where the cluster may differ is in resource capacity or nearness to a given client, but these are operator deployment choices, and an "Edge" could have just as much or more capacity as an "Upstream" cluster.

Based on the above, it is possible to qualify existing or purpose-built MEC hosts as edge clouds for placement of services required by applications that use them.

This edge computing requirement driven optimization of network and compute resources also can be achieved by re-configuration of the underlay connecting MEC hosts.

3.2.3 MEC System Level Scheduling

At the Kubernetes multi-cluster (multi-MEC host level), scheduling is provided via the KubeFed project. At the federation level, the scheduling process changes from scheduling a single pod to a node to delegating or replicating Kubernetes resources to a cluster. The constraint-based scheduling described above in the context of a single cluster can be applied at the system level with minor additions to the capabilities.

In KubeFed's existing implementation of scheduling, an operator specifies how a resource is federated across the set of member clusters. This includes the specification of the cluster as well as the cardinality of a resource assigned to that cluster. There is a capability to allow the federation to be ratio-based as opposed to completely explicit, but this still equates to a prescriptive federation.

By augmenting KubeFed's scheduling algorithm, as it does not provide the same extension mechanism that base Kubernetes provides, constraint-based scheduling, including connectivity-based constraints, can be achieved. Instead of specifying a cardinality and a cluster, the cardinality and connectivity constraints can be specified allowing the scheduler to place the workloads across the multiple clusters. After the workloads are placed, the constraints can be monitored and upon violation, the resources can be rescheduled within the currently assigned cluster or to another cluster.

3.2.4 MEC System Level Networking

Kubernetes does not provide inter-cluster networking capability natively nor as part of KubeFed project. CNCF provides multi-cluster DNS capability that can be used in a multi-cluster deployment.

Inter-cluster connectivity can be facilitated via the exposing of cluster services via a standard Kubernetes ingress controller or the NSM. Additionally, as part of the scheduling process, a network controller can be used to establish new network paths or modify existing paths.

When using an ingress controller, the services provided through a given cluster are exposed on a public IP address and port. This allows services from other clusters to access these services. The downside of this approach is that it only supports layer 3 (L3), and in some implementations only HTTP connections.

With an NSM implementation, inter-cluster networking can be established through peer to peer connections between NSM managers in each cluster. This allows the establishment of layer 2 (L2) and L3 connections. Further, using sidecars, this connectivity can be declarative, maintaining the SOC between application development and application deployment.

Where a network controller can be integrated, either through a scheduler extension or a DevOps pipeline, new network connections can be established that meet the connectivity-base constraints specified via the constraint policy system. Between the use of the network controller and the NSM complex, inter-cluster networking scenarios can be supported.

4 Summary

This document described extensions and additions to the standard Kubernetes deployment that provide constraint-base, specifically connectivity-based constraints, scheduling of Kubernetes workloads. Additionally, support for VM as well as container-based workloads was introduced, including chaining of those workloads. How these capabilities can be applied to a single Kubernetes cluster and a federation of Kubernetes clusters was described. Additionally, how these technologies could be applied to non-operator cloud capabilities (i.e., hyper-scaler Kubernetes clusters) was described.

This document then showed how the Kubernetes-based technologies could be deployed to provide an architecture that is compliant with the MEC architecture and how the components of the Kubernetes deployment map to the MEC architecture.

In summary, this document has shown how a MEC compliant multi-host system can be deployed using existing CNCF projects with a few key extensions providing a declarative based, autonomous system for MEC service deployments.

Abbreviations and Definitions

API	application programming interface
AR/MR	augmented and mixed reality
BSS/OSS	business support system / operations support system
B2B	business to business
B2C	business to consumer
CNCF	Cloud Native Computing Foundation
CNF	cloud-native network function
CNI	container networking interface
CRD	custom resource definition
CSP	communication service providers
DNS	domain name service
ETSI	European Telecommunication Standards Institute
HTTP	hypertext transfer protocol
K8s	Kubernetes
L2	layer 2 networking
L3	layer 3 networking
MEC	multi-access edge computing
NF	network function
NFV	network function virtualization
NSM	network service mesh
NFVO	network function virtualization orchestration
MANO	management and orchestration
PAAS	platform as a service
QOS	quality of service
SCTE	Society of Cable Telecommunications Engineers
SIG	special interest group
SOC	separation of concern
VM	virtual machine
VNF	virtual network function

WAN	wide area network

Mixed Reality	Merging of physical and virtual worlds to produce new environments and visualizations, where physical and digital objects co-exist and interact in real time.
Augmented reality	Related to Mixed Reality term, and it takes place in the physical world, with information or objects added virtually.
Edge Computing	The delivery of computing capabilities to the logical extremes of a network in order to improve the performance, operating cost and reliability of applications and services. By shortening the distance between devices and the cloud resources that serve them, by reducing network hops, edge computing mitigates the latency and bandwidth constraints of today's Internet, ushering in new classes of applications. In practical terms, this means distributing new resources and software stacks along the path between today's centralized data centers and the increasingly large number of devices in the field, concentrated, in particular, but not exclusively, in close proximity to the last mile network, on both the infrastructure and device sides.
Edge Cloud	Cloud-like capabilities located at the infrastructure edge, including from the user perspective access to elastically-allocated compute, data storage and network resources. Often operated as a seamless extension of a centralized public or private cloud, constructed from micro data centers deployed at the infrastructure edge. Sometimes referred to as distributed edge cloud. Implementation of these capabilities with Kubernetes clusters is this paper's focus.

References

[1] ETSI GS MEC 003 v2.2.1 (2020-12): *Multi-access Edge Computing (MEC); Framework and Reference Architecture*; European Telecommunications Standards Institute; https://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/02.02.01_60/gs_MEC003v020201p.pdf

[2] Kubernetes Documentation: Glossary (10-AUG-2021); https://kubernetes.io/docs/reference/glossary/?all=true#term-cluster

[3] Helm: project home page (10-AUG-2021); https://helm.sh/

[4] CNCF Multicluster Special Interest group (10-AUG-2021); https://github.com/kubernetes/community/tree/master/sig-multicluster

[5] CNCF Node Feature Discovery (10-AUG-2021); https://github.com/kubernetes-sigs/node-feature-discovery

[6] CNCF Descheduler (10-AUG-2021); https://github.com/kubernetes-sigs/descheduler

[7] ETSI GR NFV-IFA 029 V3.3.1 (2019-11): Network Functions Virtualisation (NFV) Release 3; Architecture; Report on the Enhancements of the NFV architecture towards "Cloud-native" and "PaaS"; European Telecommunications Standards Institute; https://www.etsi.org/deliver/etsi_gr/NFV-IFA/001_099/029/03.03.01_60/gr_NFV-IFA029v030301p.pdf

[8] ETSI GS NFV-IFA 040 V4.2.1 (2021-05): Network Functions Virtualisation (NFV) Release 4; Management and Orchestration; Requirements for service interfaces and object model for OS container management and orchestration specification; European Telecommunications Standards Institute; https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/040/04.02.01_60/gs_NFV-IFA040v040201p.pdf

[9] Global System for Mobile Communications (13-AUG-2021): 5G Operator Platform; https://www.gsma.com/futurenetworks/5g-operator-platform/

[10] Kubernetes Documentation (13-AUG-2021): https://kubernetes.io/docs/home/

[11] Mobile Experts, “Edge Computing for Enterprises” (July 2019); https://mobile-experts.net/Home/Report/1152

[12] Kubernetes Components (13-AUG-2021), https://kubernetes.io/docs/concepts/overview/components/