# Service Function Chaining Implementation using VNFs and CNFs

Abdullah Bittar
*Dept. of Systems and Computer Engineering*
Carleton University
Ottawa, Canada
abdullahbittar@cmail.carleton.ca

Ziqiang Wang
*Dept. of Systems and Computer Engineering*
Carleton University
Ottawa, Canada
ziqiangwang@cmail.carleton.ca

Changcheng Huang
*Dept. of Systems and Computer Engineering*
Carleton University
Ottawa, Canada
huang@sce.carleton.ca

***Abstract*—The increasing popularity of cloud-based platforms has led to the emergence of two prominent open-source solutions: OpenStack and Kubernetes. OpenStack facilitates computing and networking resources through virtual machine instances, while Kubernetes excels in container orchestration, managing containerized workloads and services. This paper explores the implementation of Service Function Chaining (SFC) using a combination of virtual machines in OpenStack and containers in Kubernetes. The primary focus of our study is to analyze the performance of containers within the context of chain deployment in Kubernetes. Our experimentation reveals compelling insights into container bootup times, showcasing their efficiency when compared to virtual machines. Additionally, we meticulously evaluate the impact of integrating SFC-related interfaces into pods forming a chain, particularly assessing CPU, memory, and bandwidth utilization. Our findings underline the advantages of utilizing containers in SFC deployment scenarios and shed light on the potential overhead that arises during interface integration.**

***Index Terms*—Service Function Chain, SFC, OpenStack, Kubernetes, VNF, CNF**

## I. Introduction

Network operators in enterprise networks [1], mobile networks [2], and data centers [3] often demand traffic to travel through multiple network Functions (NFs) in a specific order (e.g., firewall, IDS, proxy) [4], which is generally known as Service Function Chaining (SFC). Software-Defined Networking (SDN) enforces chaining policies by steering traffic through the appropriate NFs [4]. At the same time, Network Function Virtualization (NFV) can offer flexible and dynamic virtual network provisioning. SFC helps automate traffic flow between services while optimizing network resources to improve application performance using the best available routing path. SFC is vital in next-generation networks supporting technologies such as 5G, IoT, and edge computing [5]–[7].

Several cloud-based platforms have been developed and made accessible to the public for diverse computing needs. OpenStack and Kubernetes stand out among these platforms as two of the most robust solutions. OpenStack, a free and open-source cloud computing platform, facilitates the provisioning of virtual machines and other crucial resources to users [8]. However, the deployment of Network Functions (NFs) in the form of Virtual Network Functions (VNF) through Virtual Machines (VMs) may introduce challenges in terms of efficiency when dealing with large-scale 5G or edge deployments. These scenarios require enhanced agility, scalability, and minimized overhead. A cloud-native approach utilizing containers instead of VMs becomes imperative to address these challenges. Cloud-native Networking Functions (CNFs) represent a specialized extension of VNFs designed explicitly to operate within containers [9]. In this context, Kubernetes emerges as a prominent open-source cloud platform adept at managing containerized workloads and services [10]. By leveraging Kubernetes' container orchestration capabilities, organizations can achieve a more streamlined and resource-efficient deployment of CNFs, meeting the demands of modern cloud environments.

This paper focuses on implementing the SFC concept in both OpenStack and Kubernetes for a practical, real-world experience. Specifically, we concentrate on the performance differences between VNFs and CNFs by providing performance results extracted from Kubernetes. CNFs are lighter than VNFs, leading to faster booting-up time. But when adding interfaces related to chaining services, we observed an overhead that degraded the performance. We will cover the background on OpenStack, Kubernetes, and VMs and containers in Section II, the experiment setup in Section III, the performance results and analysis in Section IV, and related work in Section V. Finally, a conclusion is in Section VI.

## II. Service Function Chaining Implementation

This section will include details on the SFC implementation steps in OpenStack II-A and Kubernetes II-B. In II-C we provide the differences between VMs and CNFs.

### A. OpenStack

OpenStack acts as a fabric that can build a virtualized data center on-demand with minimal realization time, providing a large pool of computing, storage, and networking resources. OpenStack identifies SFC as "a mechanism for overriding the basic destination-based forwarding that is typical of IP networks" [8]. SFC can be implemented as an extension to the OpenStack networking module and makes it possible to create a traffic-steering service chain using only port names since all OpenStack networking services and compute instances connect to a virtual network via ports.

2694-0825/23/$31.00 ©2023 IEEE
DOI 10.1109/IC2E59103.2023.00029

A specific four-step logic approach exists for developing a chain in OpenStack infrastructure. The first step is to create a Flow Classifier (FC) used for traffic classification based on predefined policies. A classifier will classify incoming traffic, and if it matches the FC rules, traffic flow will be directed to an SFC. The second step is creating a Port Pair (PP), representing a service function instance's ingress and egress port. The port can be either uni- or bi-directional. The third step is to create a Port Pair Group (PPG), a collection of one or more PPs. Multiple PPs enable load balancing over a set of equivalent service functions. PPG has an interesting feature that may be used for monitoring and analyzing processes. When creating the PPG, users can add the *tap* attribute for the service function instance, where the instance will only receive a copy of the flow without forwarding it to the next hop. The last step is the Port Chain (PC) which defines and implements the Service Function Path (SFP) by identifying the set of FCs and an ordered list of PPGs. Traffic that matches the FC rules will have to traverse the PPs placed in the PPGs. A unidirectional PC applies when only the forward flows in the SFC chain must match the criteria specified in the FC. A bi-directional will have a symmetric feature where forwarding and reverse traffic will obey the requirements specified in the FC.

After a chain has been designed and deployed, traffic policies are implemented in the OpenStack network subsystem by adding OpenFlow rules to Open vSwitch (OvS)-based virtual switches in each compute node [11]. Integrating bridge `bri-int`, to which all instances running in a given physical node are connected, and the tunnelling bridge `br-tun`, from which tunnels between instances on separate physical nodes are configured with the ingress and egress traffic steering rules, respectively. If traffic must traverse multiple compute nodes, each hop included in the path must be configured correctly on the physical network infrastructure. For this purpose, packets must be encapsulated, and OpenStack only supports two types of encapsulation, Multiprotocol Label Switching (MPLS) or Network Service Header (NSH).

### *B. Kubernetes*

Kubernetes, in simple words, acts as a container orchestrator. Kubernetes is a microservice architecture, which means developing a single application as a collection of small services, each running in its process and communicating with lightweight mechanisms, typically through HTTP service/gRPC service and corresponding API.

Kubernetes, as a powerful container orchestration platform, exhibits minimal networking services, serving solely as a networking model placeholder within the cluster. To address this limitation, Kubernetes relies on third-party projects that offer network functionality. These external projects develop diverse extensions catering to various networking module requirements and objectives. Consequently, users are tasked with the responsibility of researching and selecting compatible networking extensions that support SFC. The deployment of an SFC in Kubernetes entails three essential stages: conducting an online search for third-party networking extensions that support SFC, implementing the correct configurations to enable SFC support, and finally, creating the SFC within the Kubernetes environment. This paper investigates the dependency on third-party networking extensions in Kubernetes to facilitate efficient and customized SFC deployment, contributing to the optimization of network capabilities within containerized environments.

The first stage is to search for third-party projects (network extensions) that support SFC in Kubernetes. Luckily, there are a few options available for users. We found three different network extensions that support SFC in Kubernetes. The first extension is called OVN4NFV [12]. This plugin is an Open Virtual Network (OVN) based on the Container Network Interface (CNI) [13] controller, providing cloud-native-based SFC and other overlay networking features. The second network extension is Contiv-VPP, a CNI plugin that employs a programmable CNF vSwitch based on FD.io/VP offering SFC and other high-performance cloud-native networking and services [14]. The Fast Data Project (FD.io) is a collaborative open-source project focusing on terabit software dataplane. Hence, they use the Vector Packet Processing (VPP) concept, which processes multiple packets simultaneously with low latency. The third and final network extension that supports SFC in Kubernetes is the Network Service Mesh (NSM) [15]. NSM is a novel approach to solving complicated L2/L3 use cases in Kubernetes that are tricky to solve, and one of its features is to provide the policy-driven SFC concept. In our experiment, we chose NSM as the network extension.

The second stage involves the configuration of the Kubernetes cluster to facilitate the deployment of an SFC service with the chosen network plugin. While most networking extensions share a common concept for identifying an SFC service, variations lie in the attribute values within the configuration files. To achieve this, Custom Resource Definition (CRD) proves instrumental, enabling users to define a custom resource with a specific name and schema. In our research, we established a CRD named *NetworkServiceChain*, a distinct identifier that plays a pivotal role in the subsequent step of creating and deploying a *Service* to declare the SFC. The *Service* acts as an abstract representation, defining a cohesive set of pods and the access policy to govern them within the cluster. This configuration process ensures a streamlined and consistent approach to implementing Service Function Chaining in the Kubernetes environment.

The third stage in building an SFC in Kubernetes is to deploy the pods in the Kubernetes cluster. This step is container development and encapsulates them in a pod. Developers must create containers to perform their application's service or network function. To attach a pod to a chain in Kubernetes, a metadata attribute must be added to the pod's deployment file. Our previous work [16] provides more details on deploying SFC in Kubernetes using NSM.

### *C. Virtual Machine vs. Containers*

An SFC would comprise physical or virtualized NFs that traffic has to pass through before reaching the destination.

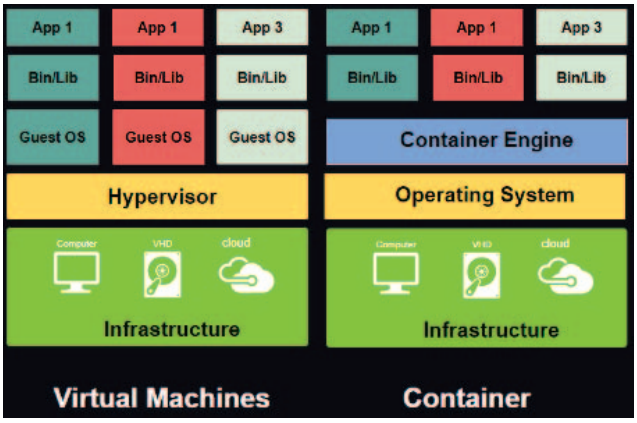


Fig. 1: Virtual Machine vs container OS architecture

The NFs can be either instantiated as a VM or container. The difference between VMs and containers is the level of Operating System (OS) virtualization.

Traditional VMs solution depends on the OS hypervisor, which manages physical computing resources and makes isolated hardware slices available for creating VMs [17]. Each VM requires a complete implementation of a guest OS, including the binaries and libraries necessary for applications, which might lead to the VM being several gigabytes in size. As a result, the guest OS will compete for resources with programs operating on the VM service, degrading QoS for the applications [18]. Contrarily, lightweight virtualization technologies such as cloud-native containers are flexible and efficient. A container shares the host OS kernel, binaries, and libraries, which come in megabytes. Containers can act as a virtualized resource [19], [20], incur significantly lower overhead than traditional Virtual Machines (VMs) [21] and have faster network speed than traditional VNF [22], even though not all virtual network functions are feasible to be containerized [23]. A Container-native Network Function (CNF) [24] is a software implementation of a network function built and deployed in a cloud-native method [25]. Despite all the benefits of integrating containers into the NFV environment, management and orchestration challenges may hinder the utilization of container-based VNFs.

If we closely look at how VMs are built over the physical hardware in Fig 1, there is a layer of Hypervisor which sits between physical hardware and operating systems. On the other hand, Containers are like normal operating system processes. They are isolated from other processes by the namespace concept. Namespace has its own isolated resources without actual partitioning of the underlying hardware. Performing SFC for containers imposes a different process than virtual machines.

## III. Experiment Setup

**Testbed** The OpenStack experiment consists of two servers. One of the servers has a 40-core CPU (Intel Xeon E5-2650 @ 2.30) with one 1GbE NIC (Intel I350), and the other server has a 40-core CPU (Intel Xeon Silver 4114 @ 2.20GHz) with one 1GbE NIC (BRCM 5720). The Kubernetes experiment was conducted on the Google Cloud Platform, consisting of three servers.

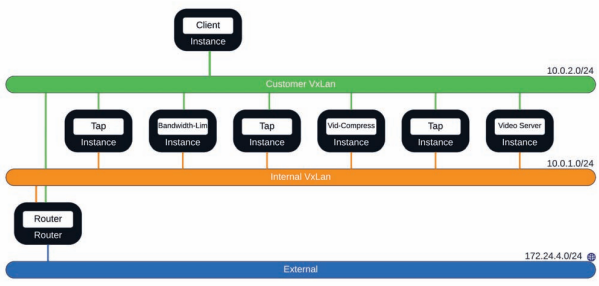


Fig. 2: Virtual network topology

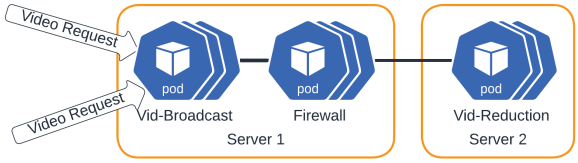


Fig. 3: SFC in Kubernetes multi-node cluster topology

**OpenStack** We used DevStack [26] to install OpenStack. Our topology consisted of seven VMs, as depicted in Fig. 2. One VM was acting as a client, another VM was working as a video storage server, two Service Functions (SFs) VMs, and three *tap* VMs. The SFs are video compression for compressing video, and the second SF is bandwidth limiter, ensuring the compressed video is below a threshold. The *tap* VMs were located across the chain for monitoring purposes to ensure the traffic was going through the overlay and not the underlay network.

**Kubernetes** We focused on using the latest version of Kubernetes. The client version for Kubernetes is 1.21.3, and the server version for Kubernetes is 1.21.1. We deployed Kubernetes on the three servers mentioned above, one server acting as a master node and two servers acting as working nodes. Our chain consists of three CNFs in a sequence, as illustrated in Fig. 3. We added a custom-build container to perform a specific function inside each pod. The functions we chose were firewall, video compression and video broadcast. We chose the NSM networking extension to be deployed in our cluster. The new release of NSM (v1.0.0) does not depend on a specific CNI. Therefore, we selected Weave Net, a resilient and straightforward network for Kubernetes and its hosted applications [27]. Weave Net creates a virtual network that connects Docker containers across multiple hosts and enables their automatic discovery.

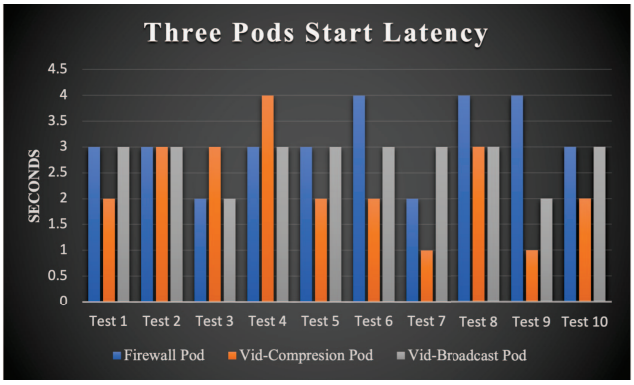


Fig. 4: Pod start latency without chain attachment for three pods

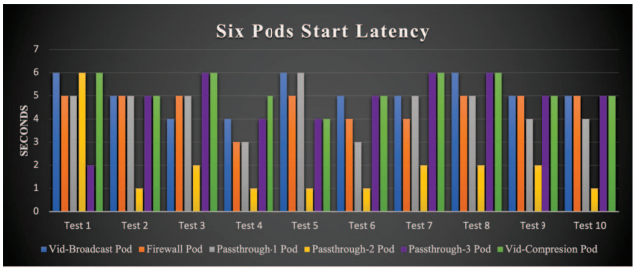


Fig. 5: Pod start latency without chain attachment for six pods

## IV. Performance Results and Analysis

Our experiment aims to create a dynamic, customized SFC with a web-based orchestrator across multiple nodes. In our previous work [16], we designed and implemented an SFC network in Kubernetes using the NSM framework. Further, we demonstrated a prototype [28] that dynamically allows users to deploy network function chains using a web interface-based orchestration. The orchestrator automates the SFC developing process, allowing users to choose different container-based microservices and define the routing rules. The system also integrates a resource and network state monitoring solution supported by the Prometheus system [29]. The monitoring framework provides metrics to optimize network performance and validate the SFC path. This study will present performance test findings related to SFC in Kubernetes. The complexity of the function itself, the relationships between processes, and many other complex elements can affect how well the SFC performs. Significant parameters that reflect the entire system's performance can be measured and gathered despite such impractical characteristics. These parameters comprise hardware conditions such as host and container CPU and memory utilization, host I/O device statuses and CPU temperature. Network performance, including the throughput and latency of the container network, is another factor in the performance evaluation for SFC implementation. Additional environmental factors would also be considered, such as service latency, HTTP request error rate, and the trend of request delays, to

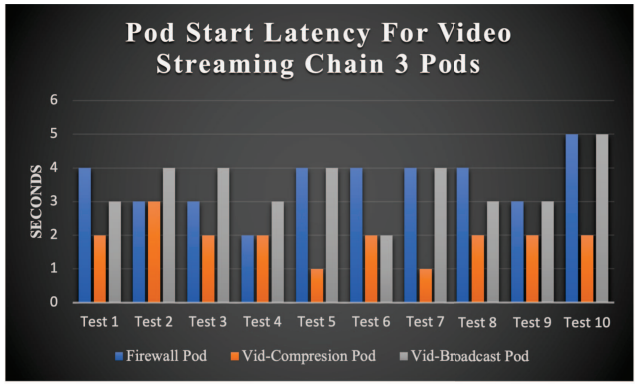


Fig. 6: Pod start latency for video streaming SFC with three pods

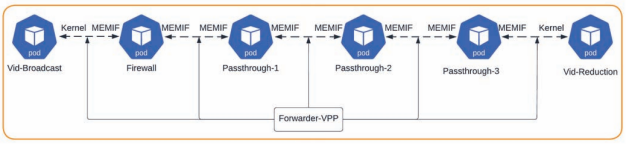


Fig. 7: Networking architecture of the 6-pod video streaming SFC deployment

cross-validate the network limits. We discuss the performance of the suggested approach in terms of SFC initialization latency IV-A, NSM control plane resource utilization IV-B, communication channel bandwidth IV-C, inter-services traffic latency IV-D, and service fault recovery time IV-E.

### A. Chain Initialization Latency

Kubernetes allows the usage of any interface the network extension deploys. The NSM network plugin uses Kernel and memif interfaces. Specifically, NSM allows the edge pod interface to be Kernel interfaces and anything between the chain only to have memif interfaces. We conducted three tests to investigate the effect of adding the NSM interface while creating the chain. Our first test measured the pod initialization time without adding the NSM interface to the pods. In other words, how long does it take to deploy a pod in the Kubernetes cluster without any NSM interfaces added to it? The initialization latency calculated in this experiment does not include the time for downloading the image from the cloud to the local container runtime. Figure 4 illustrates the results we found. We deployed three pods simultaneously and repeated the test ten times. The minimum amount of time required to deploy a pod and be in a ready status is one second, and the maximum is four seconds. The average pod initialization time for three pods without chain attachment was 2.73 seconds. We also tested the time required to deploy six pods. The average pod initialization time for six pods without chain attachment was 4.36 seconds, as illustrated in Figure 5. There is an increase of over 60% in the pod initialization latency when the number of pods doubles.

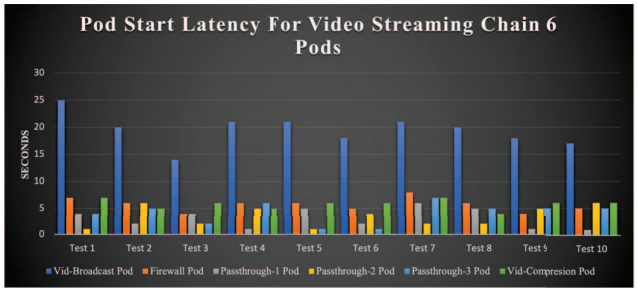


Fig. 8: Pod start latency for video streaming SFC with six pods


Fig. 10: NSM control plane components memory size usage during SFC deployment

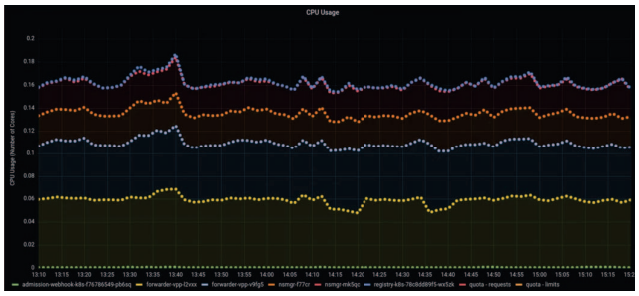


Fig. 9: NSM control plane components CPU unit usage during SFC deployment

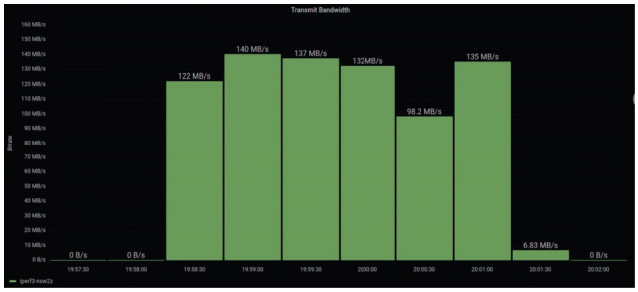


Fig. 11: The iperf3 bandwidth test result using WeaveNet interfaces visualized in Prometheus

The second test measures the pod start latency and focuses on adding these pods to a chain. We deployed our video streaming chain, which included three pods. Figure 6 illustrates the results. The minimum amount of time required to deploy a pod and be in a ready status is one second, and the maximum is five seconds. The average pod initialization time for three pods with NSM interface attached to pods was three seconds.

In the third test, we increased the chain length from three pods to six pods. We added three passthrough pods as illustrated in Figure 7. We wanted to investigate if the pod startup latency will increase if the chain length increases. As illustrated in Figure 8, we observed a significant increase in pod startup time. The pod initialization varied from one second to 25 seconds. Mainly, the Vid-broadcast pod took the longest time to be ready. The average pod initialization time for six pods with NSM interface attached to pods was 6.93 seconds. The average pod initialization increased more than double compared to three pods SFC. This is still a better result than 11.48 seconds to bootup VMs [19].

### *B. NSM Control Plane Resource Usage Performance*

In this test, we analyzed the NSM control plane components: the NSM register, NSM manager, NSM forwarder and NSM admission webhook. Since we have one master node and two working nodes, each working node will have its own NSM manager and NSM forwarder. We deployed the 3-pods video streaming chain and monitored the CPU and memory usage during the SFC creation period. Specifically, we deployed the chain and waited for 10 minutes before deleting the chain and redeploying the chain again. We repeated this test five times. In Kubernetes, 1 CPU unit is equivalent to 1 physical CPU core or one virtual core, depending on whether the node is a physical host or a virtual machine.

Figure 9 and Figure 10 illustrate the results we found of CPU and memory usage, respectively, vs. time that was collected and plotted as a time series graph in real-time using Prometheus web UI. To clarify, we deployed the video streaming chain five times, strictly at 13:30, 13:50, 14:10, 14:30, and 14:50. Furthermore, during the deployment at 14:10 and 14:30, traffic flow was traversing between the pods in the chain. From Figure 9, we can observe a steady increase in CPU usage when the chain is deployed. However, even during the traffic flow period in the chain, we don't see a drastic change in CPU usage. We observe similar results for the memory utilization for the NSM control plane components, as depicted in Figure 10. A steady increase in memory usage when the chain is deployed and a steady decrease when the chain is removed.

### *C. Bandwidth Utilization*

We used the iperf3 container image, a wildly used software tool, to measure network performance. We wanted to study the bandwidth performance of the SFC data plane supported by the NSM traffic forwarder. In this experiment, the iperf3 sender generated TCP sessions and sent them to the receiver at the other end of the chain. The TCP maximum segment size is 1460 bytes, and the TCP buffer size is 128 KB.

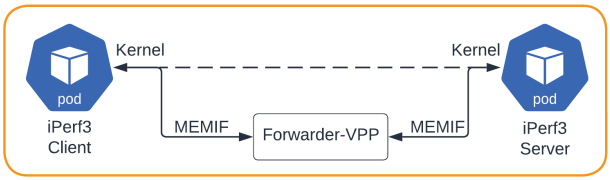


Fig. 12: Networking architecture of the bandwidth test deployment for NSM interfaces

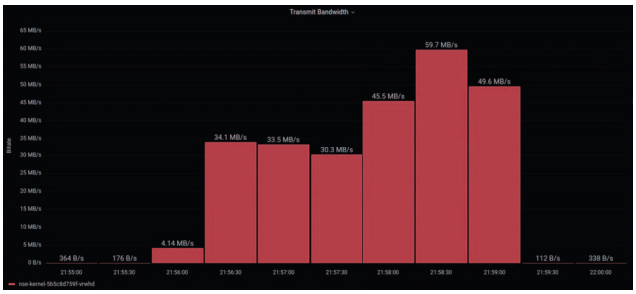


Fig. 13: The iperf3 bandwidth test result using WeaveNet interfaces visualized in Prometheus

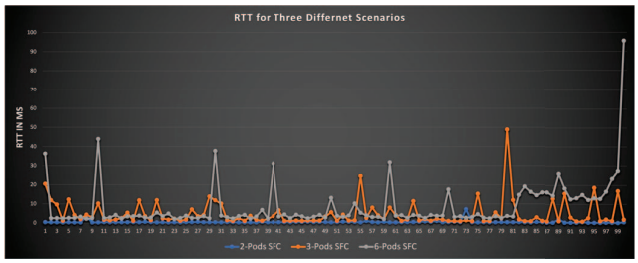


Fig. 14: The RTT test results from different container deployment cases

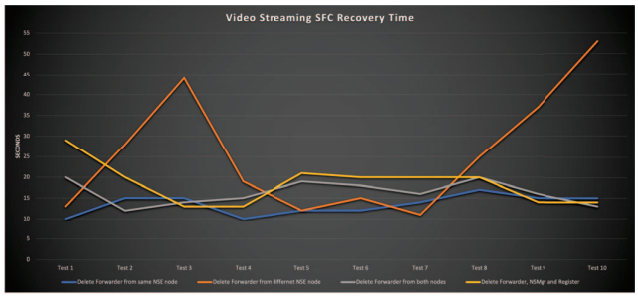


Fig. 15: SFC recovery time for four fault scenarios.

Our first experiment measures the network bandwidth between two default Linux Kernel interfaces provided by WeaveNet. This test only involved an iperf3 server (receiver) and an iperf3 client (sender). Hence it is the reference representing the bandwidth of Kubernetes' non-SFC data plane when comparing the bandwidth performance supported by the NSM interfaces (SFC data plane). We used the Prometheus monitoring system to collect real-time network bandwidth, where the average throughput is calculated relying on the PromQL built-in functions. As depicted in Figure 11, the average bandwidth of the default CNIs was 110 MBps, and the maximum bandwidth reached was 140 MBps during the test period. The x-axis is the system time, while the y-axis is the bandwidth of the iperf3 transmit interface. The unit of the y-axis is Megabytes/second (MBps).

The second experiment test only involved an iperf3 server, an iperf3 client, and the NSM traffic forwarder in measuring the maximum bandwidth provided by NSM interfaces. The pods that ran iperf3 containers established network connections using an NSM traffic forwarder to measure the maximum bandwidth between the two NSM interfaces as demonstrated in Figure 12. The iper3 bandwidth test results are depicted in Figure 13. It shows that the maximum bandwidth reached by the NSM data plane was 59.7 MBps which is less by two times than the network bandwidth supported by the WeaveNet Linux Kernel interface.

### *D. Latency Between Network Services*

High latency becomes a problem as networks get bigger since more connections mean more points of failure, where issues and delays can happen. These risks increase as end users connect to remote cloud servers and numerous network services across various domains. For the network operator to properly plan the network application, assessing the network latency between network services is crucial. The time it takes for data or a request to go from the source to the destination is known as network latency, measured in milliseconds. The Round-Trip Time (RTT) was obtained from the ICMP ping test, including echo and echo reply messages. The packet size for each echo request was set to 1500 bytes instead of the default 56 bytes to increase the traffic load.

Similar to the bandwidth experiment, we have three different tests. The first test will be for a chain that only includes two pods. The second test will be for a chain with three pods, our video streaming chain. At the same time, the last test will include our six pods chain. We designed three experiments based on the ICMP ping test to compare the network latency for three use cases. In each experiment, we sent one hundred ICMP packets between endpoints. Figure 14 illustrates the results we collected. The y-axis is the RTT in milliseconds, and the x-axis is the trace of one hundred packets.

**2-Pods SFC**: we created a two-pod chain, one client pod as a sender and one destination pod as the receiver, while traffic traverses through the NSM traffic forwarder. This test is considered the baseline latency between two pods which adopt the NSM CNIs. The test results are illustrated by the blue line in Figure 14 with an average RTT time of 0.529 ms.

**3-Pods SFC**: we deployed our video streaming chain that includes three pods. The test results are illustrated by the orange line in Figure 14 with an average RTT time of 4.872 ms.

**6-Pods SFC**: we increased the chain length by adding three passthrough pods as illustrated in Figure 7. The test results are

illustrated by the orange line in Figure 14 with an average RTT time of 8.402 ms.

### *E. NSM Reliability*

Fault detection for SFC is managed as the health monitoring for pods and virtual links between pods. The recovery includes two parts: pod recovery and SFC re-configuration. The healthiness and liveness probes used by the Kubernetes cluster detect the pod's failure and schedule a new pod to replace the faulty container-based NSE while the NSM control plane re-configures the SFC connections.

The fault in this test was simulated by manually deleting one or more components consumed by an SFC. The ICMP ping test was used to check the connectivity of the SFC path. Specifically, we chose the video reduction pod, which we will call for now ping pod, to ping the video broadcast pod and deploy the fault scenario. The faulty components included NSE (e.g., firewall and video compression pod), NSMgr, NSM Registry, VPP forwarder, or any combination. We selected four fault scenarios to test the SFC framework's reliability. The four scenarios involve:

1) Deleting the NSM forwarder from the same ping pod node;
2) Deleting the NSM forwarder, not on the same ping pod node;
3) Delete NSM forwarder from both nodes; and
4) Delete NSM forwarder, NSMgr and Register

For each fault scenario, ten tests were conducted to collect and calculate the average SFC path recovery time. Figure 15 shows the SFC path recovery time for all four fault scenarios. The y-axis is the fault recovery time in seconds, and the x-axis is the trace of ten tests.

Table I illustrates the recovery time against different fault scenarios. As shown in Table I, the most prolonged service downtime occurred when deleting the forwarder that was not on the same node as the ping pod. When deleting this forwarder, in some cases, as indicated in the table with red font, the other forwarder on the other node is forced to be re-deployed. Furthermore, the traffic forwarder took most of the time reconnecting the data plane elements during the SFC path recovery process. The average recovery time was 25.7 seconds for traffic forwarder restart. Meanwhile, the average recovery time for deleting NSM forwarder, the NSMgr and NSM Registry recovery took 18.4 seconds.

### *F. Analysis*

The length of the SFC does not significantly affect the initialization latency of the SFC, according to the results in Table II, as long as the container runtime caches all the container images used by the network services. We observed increased pod start-up time for the Vid-broadcast pod in the 6-pods SFC. The Vid-broadcast pod acts as the client pod under the NSM architecture. Deploying the client pod was the most time-consuming part of deploying an SFC because the client pod must wait until the rest of the network service endpoints are registered with the NSMgr and deployed in the cluster, then the chaining process starts by deploying the client pod. The bandwidth experiment test was to see if there is any network overhead when adding the NSM interface to containers when creating an SFC. Depending on our results in IV-C, we observed a drop of more than half in average bandwidth when adding the NSM interfaces. We can identify that NSM interfaces injected into containers creates network overhead.

The latency results in Table III provide the minimum, maximum and average time for all three use cases test. There is an increase of eight folds between the 2-pods and 3-pods average test. We should observe an increase in the RTT between the 2-pods and 3-pods, as traffic will have to go through a firewall before reaching the destination. However, a considerable overhead is added to the network when increasing the chain from two to three pods. Furthermore, we also observed an increase in the RTT between the 3-pods and 6-pods tests by almost two folds. Again, the increase in RTT is inevitable as we added three more passthrough pods to the chain. These test results demonstrate that the network latency incrementation is close to a linear relationship with the number of elements in the SFC path.

Another point to mention is the variation of the RTT in each test. For the 2-pods scenario, the minimum RTT time was 0.166 ms, and the maximum RTT was 7.143 ms, as illustrated in Table III. For the 3-pods scenario, the minimum RTT time was 0.931 ms, and the maximum RTT was 48.843 ms. Finally, in the 6-pods scenario, the minimum RTT time was 2.066 ms, and the maximum RTT was 95.606 ms. These test results also show that the RTT becomes unstable when the length of the SFC path increases.

## V. Related Work

Many previous works study both OpenStack and Kubernetes when deploying SFC. However, we couldn't find any work that compares OpenStack with Kubernetes specifically for SFC implementation capabilities in both platforms. Below are some works that compare containers versus VMs.

Chae et al. [30] made a performance comparison between Linux containers and KVM virtual machines by running between one and four different instances with a focus on CPU, memory and disk I/O. They both had about the same CPU usage in idle mode, with the VMs having slightly less average CPU idle but with more deviation. The result from the memory comparison showed that the virtual machines used up between 3.6 and 4.6 times more memory than the Linux containers. The authors in [31] found that containers provide an overall better performance than virtual machines regarding CPU and memory in a big data environment by using Spark jobs. They also found that containers perform better from a scalable perspective, where they compared the performance of between 2 and up to 512 different containers and virtual machines. Kyoung-Tack et al. [19] build their own clouds and perform a comparison experiment between their container cloud (Docker) and their virtual machines cloud (OpenStack). The test results showed the average bootup time for the

TABLE I: The recovery time against different fault scenarios

| SFC Recover time | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Test Type** | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | **Test 6** | **Test 7** | **Test 8** | **Test 9** | **Test 10** | **Avg** |
| Delete forwarder from same NSE node | 10 | 15 | 15 | 10 | 12 | 12 | 14 | 17 | 15 | 15 | 13.5 |
| Delete forwarder from different NSE node | 13 | 28 | 44 | 19 | 12 | 15 | 11 | 25 | 37 | 53 | 25.7 |
| Delete forwarder from both nodes | 20 | 12 | 14 | 15 | 19 | 18 | 16 | 20 | 16 | 13 | 16.3 |
| Delete forwarder, NSMgr, and Register | 29 | 20 | 13 | 13 | 21 | 20 | 20 | 20 | 14 | 14 | 18.4 |

TABLE II: Pod start latency min, max and average results in seconds against different scenarios

| Use-case | Minimum | Maximum | Average |
|---|---|---|---|
| **3-Pods** | 1 | 4 | 2.73 |
| **6-Pods** | 1 | 6 | 4.36 |
| **3-Pods SFC** | 1 | 5 | 3 |
| **6-Pods SFC** | 1 | 25 | 6.93 |

TABLE III: The RTT minimum, maximum and average results in milliseconds against different scenarios

| Use-case | Minimum | Maximum | Average |
|---|---|---|---|
| **2-Pods SFC** | 0.166 | 7.143 | 0.529 |
| **3-Pods SFC** | 0.931 | 48.843 | 4.872 |
| **6-Pods SFC** | 2.066 | 95.606 | 8.402 |

containers was around 1.53 seconds, and the average time for the VMs was 11.48 seconds. In another work, Vestman [32] runs an experiment and compares the performance of virtual machines deployed through OpenStack and containers deployed through Docker regarding CPU operations, primary memory usage, disk read/write, and file transfer speed between host and application. The authors concluded that the VM handled requests faster, while the containers performed better in file transfer and resource usage. All the papers mentioned above compare the chain performance in the network either in OpenStack or Docker cloud, which is different than our work. We compare OpenStack and Kubernetes under the scoop of platform SFC capabilities.

## VI. Conclusion

We implemented SFC using VNFs in OpenStack and CNFs in Kubernetes. We further provide performance results for CNFs deployment while creating chains. We observed an increase in pod start latency as we increased the pods in the chain. The average time to bootup VMs is 11.48 seconds, while a six-pod chain average time was around seven seconds. There was a steady increase in CPU and memory usage when the chain was deployed using NSM. Furthermore, when adding the NSM interface, the bandwidth of the chain between containers decreased by two times that of the network bandwidth without the SFC interfaces. Although pods are lighter and VMs, NSM adds its own overhead. This overhead is due to the connection added between the pods and the NSM forwarder-VPP. Additionally, the NSM injects Kernel interfaces to the edge pods and memif interfaces in the middle pods. Limiting a specific interface to be injected in a pod has its ramifications. Developers will be limited to developing containers compatible with adding a memif interface alongside the container function.

## References


[1] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336, 2012.

[2] M Stiemerling D Lopez W Haeffner, J Napper and J Uttaro. *Service functionchaining use cases in mobile networks. draft-ietf-sfc-use-case-mobility-09*. IETF, 2019.

[3] Surendra Kumar, Mudassir Tufail, Sumandra Majee, Claudiu Captari, and Shunsuke Homma. Service function chaining use cases in data centers. *IETF SFC WG*, 10, 2015.

[4] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 511–524, 2016.

[5] Hajar Hantouti, Nabil Benamar, and Tarik Taleb. Service function chaining in 5g beyond networks: Challenges and open research issues. *IEEE Network*, 34(4):320–327, 2020.

[6] Yicen Liu, Yu Lu, Xi Li, Zhigang Yao, and Donghao Zhao. On dynamic service function chain reconfiguration in iot networks. *IEEE Internet of Things Journal*, 7(11):10969–10984, 2020.

[7] Abderrahime Filali, Amine Abouaomar, Soumaya Cherkaoui, Abdellatif Kobbane, and Mohsen Guizani. Multi-access edge computing: A survey. *IEEE Access*, 8:197017–197046, 2020.

[8] Openstack. Build the future of open infrastructure. https://www.openstack.org/. [Online; accessed 10-April-2023].

[9] Jane Shen and Jeff Brower. Access and edge network architecture and management. *Future Networks, Services and Management: Underlay and Overlay, Edge, Applications, Slicing, Cloud, Space, AI/ML, and Quantum Computing*, pages 157–183, 2021.

[10] Kubernetes. Production-grade container orchestration. https://kubernetes.io/. [Online; accessed 10-April-2023].

[11] Gianluca Davoli, Walter Cerroni, Chiara Contoli, Francesco Foresta, and Franco Callegati. Implementation of service function chaining control plane through openflow. In *2017 IEEE conference on network function virtualization and software defined networks (NFV-SDN)*, pages 1–4. IEEE, 2017.

[12] OPNFV. Opnfv/ovn4nfv-k8s-plugin. https://github.com/opnfv/ovn4nfv-k8s-plugin. [Online; accessed 10-April-2023].

[13] CNI. Cni - the container network interface. https://www.cni.dev/. [Online; accessed 10-April-2023].

[14] ContiVPP. Contivpp.io. https://contivpp.io/. [Online; accessed 10-April-2023].

[15] Network Service Mesh. The hybrid/multi-cloud ip service mesh. https://networkservicemesh.io/. [Online; accessed 10-April-2023].

[16] Abdullah Bittar, Ziqiang Wang, Amir Aghasharif, Changcheng Huang, Gauravdeep Shami, Marc Lyonnais, and Rodney Wilson. Service function chaining design & implementation using network service mesh in kubernetes. In *Asian Conference on Supercomputing Frontiers*, pages 121–140. Springer, Cham, 2022.

[17] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[18] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[19] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, 66(105-111):2, 2014.

[20] Blesson Varghese, Lawan Thamsuhang Subba, Long Thai, and Adam Barker. Container-based cloud virtual machine benchmarking. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 192–201. IEEE, 2016.

[21] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei Zhou. A comparative study of containers and virtual machines in big data environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 178–185. IEEE, 2018.

[22] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[23] Richard Cziva and Dimitrios P Pezaros. Container network functions: Bringing nfv to the network edge. *IEEE Communications Magazine*, 55(6):24–31, 2017.

[24] Cloud-Native Network Functions. Cloud-native network functions. https://cdnf.io/. [Online; accessed 10-April-2023].

[25] Yong-Xuan Huang and Jerry Chou. Evaluations of network performance enhancement on cloud-native network function. In *Proceedings of the 2021 on Systems and Network Telemetry and Analytics*, SNTA '21, pages 3–8, New York, NY, USA, 2020. Association for Computing Machinery.

[26] OpenDev. Openstack/networking-sfc. https://opendev.org/openstack/networking-sfc. [Online; accessed 10-April-2023].

[27] Weave Works. Integrating kubernetes via the addon, 06 2022.

[28] Ziqiang Wang, Abdullah Bittar, Changcheng Huang, Chung-Hong Lung, and Gauravdeep Shami. A web-based orchestrator for dynamic service function chaining development with kubernetes. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 234–236. IEEE, 2022.

[29] Prometheus. From metrics to insight. https://www.prometheus.io. [Online; accessed 10-April-2023].

[30] MinSu Chae, HwaMin Lee, and Kiyeol Lee. A performance comparison of linux containers and virtual machines using docker and kvm. *Cluster Computing*, 22(Suppl 1):1765–1775, 2019.

[31] Sogand Shirinbab, Lars Lundberg, and Emiliano Casalicchio. Performance evaluation of container and virtual machine running cassandra workload. In *2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*, pages 1–8. IEEE, 2017.

[32] Simon Vestman. Cloud application platform-virtualization vs containerization: A comparison between application containers and virtual machines, 2017.