Layer-Aware Containerized Service Orchestration in Edge Networks

Mahdi Dolati, Seyed Hamed Rastegar, Ahmad Khonsari, and Majid Ghaderi

***Abstract*—Edge computing provides computational resources in the vicinity of end-users to reduce delay compared to traditional remote clouds. However, the capacity of edge resources usually is not sufficient for the required computational demands. Therefore, it is necessary to design methods for employing these resources in an efficient manner. On the other hand, network function virtualization (NFV) is a promising solution to use the network resources in a more flexible way than traditional schemes. Although more focus has been on realization of NFV systems via virtual machines so far, recent studies show that container-based solutions can improve efficiency thanks to lightweight implementation and layered structure of containers. Nonetheless, to the best of our knowledge, there is no comprehensive study on the problem of orchestrating services composed of a chain of containerized network functions in edge networks. In this paper, we consider this scenario when service requests are submitted to the system and address important aspects of this problem such as downloading and sharing container layers and steering traffic among network functions. We present the formulation of the problem as an integer linear program (ILP) and prove its NP-hardness. Then, to handle this problem, we propose RCCO, a polynomial-time algorithm based on ideas from deterministic and randomized rounding framework. Our results from extensive evaluations show that the bandwidth consumption of the proposed algorithm compared to the optimal algorithm is higher by only about 4% while it can outperform baselines from literature by more than 37%.**

***Index Terms*—Containerized Network Function, Edge Computing, Layer Sharing, Popularity-aware Orchestration, Service Function Chaining, Virtualization**

I. Introduction

Emerging data-intensive applications such as the Internet of Things (IoT), virtual/augmented reality (VR/AR), 6G, smart city, healthcare, and autonomous vehicles usually require considerable computation to work in the right way. However, the devices running these applications are not capable of handling all the required computations and thus it is necessary to offload them to other entities with sufficient computational resources. Traditionally, cloud infrastructure has been the only option for delivering this category of services. However, the stringent delay requirement of modern applications has challenged this deployment strategy in recent years. Edge-computing that takes computation facilities to the vicinity of the end-users is an architecture for enabling delay-sensitive and delay-critical applications and hence is an attractive solution to the above challenge [1]. According to its interesting features, the topic of edge computing recently has been the subject of many academic and industrial projects [2]. Nonetheless, due to the limited amount of available edge resources, increasing the efficiency of their usage is of a paramount importance.

Network Function Virtualization (NFV) is a prominent paradigm for efficient allocation of computing and network resources, and thus it is a viable approach for edge networks. NFV virtualizes the physical resources to share them among various virtual network functions (VNFs) and increases resource utilization. Specifically, each VNF is a special-purpose application that is isolated inside a virtual environment and can run on general-purpose hardware. Traffic is sent through a set of VNFs in a specific order to provide a customized service. The mapping of VNFs to physical resources and steering traffic among them, which is referred to as service function chaining (SFC), is a crucial procedure that determines achievable benefits of NFV. In this regard, service orchestration is defined as the process of allocating sufficient resources to VNFs, steering traffic between them, and managing all the system aspects required for providing an intended service in its lifecycle duration to the users of the network. The entity in the charge of this process is called the orchestrator.

An important factor that has a vital role in NFV is the virtualization technique. Emulating the hardware with virtual machines (VM) has been the traditional technique for virtualization of network functions in NFV. However, this technique requires the installation of an operating system along with all software dependencies that leads to a significant overhead [3] which is more concerning as each VNF provides a single application. On the other hand, container technology [4] is an attractive alternative virtualization paradigm that does not require a complete operating system installation and significantly lowers the storage overhead [5], [6]. Moreover, layering is a property of container technology that allows different containers to share and reuse executables, object files, libraries, source codes, and scripts, which further lowers the storage and bandwidth requirements of containerazed applications [7]. According to a recent study on the Docker Hub registry [8], storage space utilization can be reduced by 50% through sharing the common layers between different containers built for similar applications. Due to these advantages, the adoption rate of the container technology by organizations is witnessing a significant growth [9].

The containerized resource allocation and container layer sharing problems have been explored in the literature recently [7], [10], [11]. However, to the best of our knowledge, none of them take the chaining of functions into account. Consequently, these works that mainly focus on single service orchestration can not provide the efficiency and agility through incorporating unique features of containerization in VNF chaining. To fill this gap, we address the problem of orchestrating chains of containerized VNFs by considering

important aspects such as layered structure of VNFs, the necessity of downloading these layers for instantiating the VNFs, and the possibility of sharing layers among VNFs. Furthermore, we focus on edge-enabled networks and thoroughly capture the essential parameters to carefully model a scenario in which service requests arrive at the system over time and the network orchestrator is responsible to appropriately schedule downloading of layers from repositories and determining the right nodes to place the layers considering various network resources including link bandwidths, nodes' storage, and CPU and RAM capacities while preserving the target end-to-end delays. Our results show that sharing and reusing the container layers can significantly improve the resource efficiency. For a container image with a fixed volume, increasing the number of its layers results in higher performance gain and better resource utilization in the network. It is worth noting that our proposed scheme can be realized in practice as a module in container management systems such as Kubernetes [12]. To summarize, our main contributions are as follows:

- In this paper, we consider the problem of container layer placement and management for service function chaining in edge networks. To the best of our knowledge, this is the first work that addresses this problem.
- We carefully formulate a realistic scenario for the problem in which service requests arrive at the system over time and an orchestrator is responsible to appropriately schedule the download and placement of the layers considering various network resources including link bandwidths, node storage, CPU, and RAM capacities.
- We assume that each service function in the chain is implemented as a container. In this regard, the network orchestrator smartly employs layer sharing between container images.
- We formulate the problem as a mixed-integer linear program and prove its NP-hardness. For the resulted problem which is not tractable, we design an efficient and iterative algorithm. Moreover, our algorithm takes into account the limited capacity of edge servers and the popularity of layers to maximize the reuse of downloaded container layers and lower the network overhead.
- Through extensive simulation experiments, we evaluate the performance of the proposed scheme in terms of different important metrics such and bandwidth consumption and accept rate.

The paper is organized as follows. In section II, we review the related work and background. Section III introduces the system model. We present the problem formulation in Section IV. Proposed algorithms are explained in Section V followed by performance evaluation in Section VI. Finally, Section VII concludes the paper.

II. Related Work

In this section, we present a review of major related works in three categories. In Subsection II-A, we discuss works that do not consider container layering into account and mainly demonstrate the benefits of the lightweight virtualization of containers compared to virtual machine-based techniques. In Subsection II-B, we explore the works that go into details of layer management and sharing in their resource allocation schemes. However, different from our work, none of them address the joint problem of placement and chaining of containerized services. Finally, in Subsection II-C, we present a discussion about recent works on chaining the network services. However, none of these works consider the effect of downloading the required container layers or sharing the common layers for deploying containerized services on the bandwidth of the network. Different from all the reviewed works, we address the problem of container layer download, placement, and chaining taking into account sharing the common ones. Table I summarizes key differences of our work in comparison with reviewed works and shows the position of our work in the literature.

TABLE I: Selected Related Work

Reference	Edge	Container	Chain	Layering
[21]–[27]	✗	✗	✓	✗
[19]	✗	✓	✗	✗
[15]–[18]	✓	✓	✗	✗
[20], [28]	✗	✓	✓	✗
[29]–[31]	✓	✗	✓	✗
[7], [10], [11], [32]	✓	✓	✗	✓
This Paper	✓	✓	✓	✓

A. Containerized Layer-Agnostic Works

Various researches on containerized systems have a significant focus on implementing testbeds and experimental evaluations to show the benefits of using containers [13], [14]. In [15], FLEDGE is presented as a Kubernetes-compatible [12] container orchestrator based on Virtual Kubelets, aimed primarily at container orchestration on low-resource edge devices. Authors in [16] formulate containerized edge computing in blockchain-based Internet of vehicles (IoV) using models from queuing theory. In [17], the authors first develop a dynamic M/D/1 queuing model to analyze the end-to-end delay of the data packets of a container service flow, and then propose a delay-sensitive algorithm to solve their considered Edge container resource allocation problem. On the other hand, the work in [18] considers task scheduling in edge computing nodes with multiple processors taking into account inter-container communications. Authors in [19] considers Zipf-like distribution for the network traffic between containers. Their method employs two algorithms to divide containers of applications into blocks, and place these blocks across VMs, which is shown to reduce network traffic and latency. Also, the work in [20] considers container-based clouds and proposes a new workflow allocation approach, named GMTA, to optimize large-scale scientific workflow execution.

B. Layer-Aware Chaining-Agnostic Works

The work in [10] studies the placement of services on the storage edge nodes to maximize the satisfied demands with specified storage and delay requirements considering common container layers between different services. Authors in [11] propose protocols for live migration of edge services for mobile users by exploiting two benefits of containers including the possibility of lightweight migration compared

to VMs and their layered structured by not migrating the common layers. The work in [7] formulates a layer-aware microservice placement and request scheduling at the edge and proposes an iterative greedy algorithm to solve it. A layer sharing microservice deployment and image pulling strategy which explores the advantage of layer sharing to speedup microservice startup and lower image storage consumption is proposed in [32]. They then introduce a distributed algorithm to solve the resulting problem. Authors in [8] analyze over 167 TB of uncompressed Docker Hub images, characterize them using multiple metrics and evaluate the potential of file-level deduplication considering common layers between images. For example, they conclude that only 3% of the files in images are unique while others are redundant file copies, which means file-level deduplication has a great potential to save storage space.

C. Container-Agnostic SFC Works

Authors in [22] formulate the online virtual function mapping and scheduling problem, and proposes a set of algorithms for solving it. Particularly, they propose three greedy algorithms and a tabu search-based heuristic. Authors in [26] focus on the service rate control problem in the scheduling of SFC requests. The work in [27] presents methods to solve the routing and placement problems for SFC in an adaptive manner while preserving service deadlines. In [31], efficient algorithms for placement and chaining of VNFs in a system that enables the deployment of IoT services across multiple edges and clouds are presented. The work in [21] introduces a primary and backup VNF placement model to avoid service interruptions due to unavailability of nodes by using backup functions. Authors in [29] considers a multi-objective SFC placement problem taking into account various quality of service requirements, and avoiding edge resource congestion. To realize gradual transition to SDN in practice, authors in [24] consider a hybrid SDN to implement SFC provisioning. They formulate the problem for jointly optimizing SDN deployment and VNF placement as an integer linear program (ILP) and solve it by employing different optimization techniques. The work in [30] proposes an algorithm based on deep reinforcement learning to optimize bandwidth allocations as well as to adjust the network usage to minimize migration overhead of slice represented as an SFC. Also, authors in [28] study and report the benefits of employing container over VM in service function chaining.

III. System Model

The high-level architecture of the considered system is illustrated in Fig. 1. In this system, an orchestrator configures resources at the cloud and network edge to provide the requested service for incoming traffic that enters the system through base stations. In the following, we explain the modelling of different aspects of the edge network, cloud, and service demands. In Section V, we will present the algorithm used by the orchestrator. We adopt a time slotted model where time is discretized into successive slots. Without the loss of generality, we assume that inputs are normalized such that the length of a time slot is one. We use $\mathcal{T}$ to show the set of all time slots and accordingly variables and parameters are indexed by the time slot. Important notations are listed in Table II.

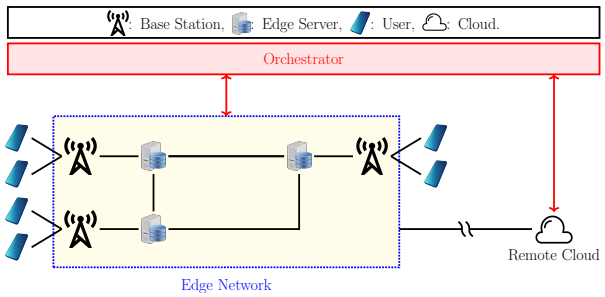


Fig. 1: Example of an edge-enabled network architecture.

Network Model. We consider a network consisting of three types of entities, namely edge servers, base stations, and a remote cloud denoted by, respectively, $\mathcal{E}$, $\mathcal{B}$, and Θ. We use $\mathcal{L}$ to show the set of links connecting the pairs of network entities in the set $\mathcal{E} \cup \mathcal{B} \cup \{\Theta\}$. The bandwidth of link $\ell \in \mathcal{L}$ is denoted by b_ℓ while d_ℓ expresses its propagation delay. We assume that the set of paths between two network entities $n, m \in \mathcal{E} \cup \mathcal{B} \cup \{\Theta\}$ is represented by $\mathcal{P}_n^m$ and the set of links along the path p are shown by $\mathcal{L}_p$. We use $\mathcal{P}$ to refer to the set of all paths between all pairs of network entities. In this network, base stations do not provide computational capacity. The cloud and edge servers provide service processing capabilities and we use $\mathcal{N} = \mathcal{E} \cup \{\Theta\}$ to collectively show them. Each edge node $e \in \mathcal{E}$ has C_e processing cores, A_e units of random-access memory (RAM), and D_e units of disk storage. The cloud is supposed to have unlimited processing, memory, and storage capacity [7].

Containerized VNF Model. We consider an NFV-based system in which service functions are realized via VNFs. We assume that VNFs, *e.g.*, access control, authentication, firewall, and image compressor, are implemented with the container technology [33]. Thus, to instantiate a VNF, we need several container layers that each layer is corresponding to a modification of the file system such as installing an application or adding a file [34]. We assume that all container layers are stored in the cloud and can be downloaded to edge servers on-demand for deploying a VNF. The downloaded layers can be deleted after service tear down to release the allocated storage space. We use $\mathcal{R}$ to show the set of all possible container layers and assume that to download layer $r \in \mathcal{R}$ we need to allocate δ_r units of storage.

Service Request Model. The service requests are submitted by users to the system over time. To deliver a service, a chain of ordered VNFs should be executed, and thus each service request is equivalent to a chain of VNFs. We use $\mathcal{U}$ to show the set of all chains that arrive to the system. For each chain $u \in \mathcal{U}$, $O_u \in \mathcal{B}$ shows the entry point, *i.e.*, the base station, from which traffic of chain u is received by the network and to which the response should be delivered in order to be passed to user. λ_u shows the expected input traffic rate of chain u. Also, τ_u shows the arrival time slot, and $\check{\tau}_u$ and $\hat{\tau}_u$ mark the start and finish time slots of the chain operation, respectively. For simplicity, we define the following two sets for each chain

TABLE II: Summary of Notations

Notation	Description
$\mathcal{U}$	Set of all service chain requests
$\mathcal{E}$	Set of all edge servers
Θ	Remote cloud
$\mathcal{B}$	Set of all base stations
$\mathcal{N}$	Set of all edge nodes and the cloud
$\mathcal{L}$	Set of all links of the network
$\mathcal{P}_m^n$	Set of all paths between network entities m and n
$\mathcal{L}_p$	Set of all links in path p
C_e	Processing capacity of edge node e
A_e	Memory capacity of edge node e
D_e	Storage capacity of edge node e
b_ℓ	Bandwidth of link ℓ
d_ℓ	Propagation delay of link ℓ
$\mathcal{R}$	Set of All container layers
$\mathcal{R}_u^i$	Container layers of VNF i in chain u
$Q_{u,e}^i$	Missing layers for placing VNF i of chain u on the edge server e at the moment of its arrival
S_u	Number of VNFs in the chain of request u
O_u	Entry point of chain u
$[n]$	Set of natural numbers from 1 to n
$\|.\|$	Total number of elements in a set, *i.e.*, its cardinality
$\tau_u, \check{\tau}_u, \hat{\tau}_u$	arrival time, the start and finish times of chain u
β_u^i	Ratio of output/input traffic rate for i-th VNF of chain u
λ_u	Expected traffic rate of chain u
λ_u^i	Input traffic rate to the i-th VNF in chain u

u to show the time slots before and after their service start:

$$\check{\mathcal{T}}_u = \{\tau_u, \dots, \check{\tau}_u - 1\}, \tag{1}$$

$$\hat{\mathcal{T}}_u = \{\check{\tau}_u, \dots, \hat{\tau}_u\}. \tag{2}$$

Notation S_u denotes the number of VNFs in service chain u that should process the traffic in a sequential manner. Also, the set of required container layers for instantiation of the i-th VNF in chain u is denoted by $\mathcal{R}_u^i \subseteq \mathcal{R}$. Since the operation of some VNFs such as image compression affect the traffic rate, for each chain u $\beta_u^i \geq 0$ represents the ratio of the output traffic rate to the input traffic rate in the i-th VNF [35]. The amount of processing power (*e.g.*, measured in GHz) and random-access memory required to process the incoming traffic at the i-th VNF are represented by π_u^i and α_u^i, respectively. We assume that each chain receives its expected processing and bandwidth resources completely. Thus, the delay from these components is deterministic and known to the service requester. The only unknown delay component is the end-to-end propagation delay. Thus, each chain specifies a threshold for the total propagation delay. We denote the maximum tolerable propagation delay of chain $u \in \mathcal{U}$ by Φ_u. However, we discuss in Subsection IV-D that we can take into account other types of delay (such processing).

IV. Problem Formulation

In this section, we present the formulation of the Containerized Chain Orchestration (CCO) problem. We formulate different parts of the problem in the following subsections. We determine the assignment of VNFs to the network entities in Subsection IV-A. Then, we formulate the download path and placement of container layers in Subsection IV-B. Then, we consider the chaining of VNFs in the process of routing the traffic in the network in Subsection IV-C and enforce the end-to-end propagation delay requirements in subsection IV-D.

TABLE III: Notable formulation notations

Symbol	Description
$v_{u,n}^i$	If the i-th VNF of chain u is in the node n
$y_{e,r}^t$	Availability of layer r in edge node e at time t
$w_{p,r}^u$	Usage of path p to download layer r by chain u
$q_{u,p}^i$	If the traffic toward the i-th VNF of chain u uses path p

Finally, we formulate the objective in subsection IV-E. Also, notable formulation variables are indicated in Table III.

A. Placement of VNFs

Each VNF of a chain should be placed in a network entity with sufficient computing, memory, and storage resources. We define binary decision variable $v_{u,n}^i$ to show whether or not i-th VNF of chain u is placed in the network entity $n \in \mathcal{N}$. The following constraint ensures that each VNF in a chain is placed in exactly one network entity:

$$\sum_{n \in \mathcal{N}} v_{u,n}^i = 1, \qquad u \in \mathcal{U}, i \in [S_u]. \tag{3}$$

The following constraints ensure that the CPU and memory capacity of edge servers are respected in every time slot:

$$\sum_{u \in \mathcal{U}} \sum_{i \in [S_u]: t \in \hat{\mathcal{T}}_u} v_{u,e}^i \pi_u^i \leq C_e, \qquad t \in \mathcal{T}, e \in \mathcal{E} \tag{4}$$

$$\sum_{u \in \mathcal{U}} \sum_{i \in [S_u]: t \in \hat{\mathcal{T}}_u} v_{u,e}^i \alpha_u^i \leq A_e, \qquad t \in \mathcal{T}, e \in \mathcal{E} \tag{5}$$

We will present the storage constraints in the next subsection. Also, notice that we do not impose storage, CPU and memory capacity constraints for the remote cloud, as we assume that its available capacity is unlimited, compared to the limited capacity of edge servers.

B. Placement of Layers

We define the binary decision variable $y_e^{r,t} \in \{0, 1\}$ to indicate whether or not layer r is available in time slot t in edge server e. We assume that the storage of all edge servers $e \in \mathcal{E}$ are empty in time slot $t = 0$, *i.e.*,

$$y_e^{r,0} = 0. \qquad e \in \mathcal{E}, r \in \mathcal{R} \tag{6}$$

We can place a VNF in an edge server only after all of its required container layers are available in that server. We write the following constraint to ensure that this requirement is respected for placement of VNFs:

$$v_{u,e}^i \leq y_{e,r}^t, \qquad e \in \mathcal{E}, u \in \mathcal{U}, i \in [S_u], r \in \mathcal{R}_u^i, t \in \hat{\mathcal{T}}_u \tag{7}$$

Our model allows VNFs to share layers. Thus, VNFs of a chain can reuse a layer available in the server that has been previously downloaded by another chain. However, an edge server can not cache downloaded layers indefinitely due to storage limitation, and thus some unused layers might be deleted. Once a layer in a server is deleted, it should be downloaded again if in the future another VNF of a chain requires the layer in that edge server. We define binary decision variable $w_{p,r}^u$ that is equal to one if download of layer r over path p is started due to the demand of chain u. Specifically, $w_{p,r}^u = 1$ implies that the download of layer r starts in τ_u

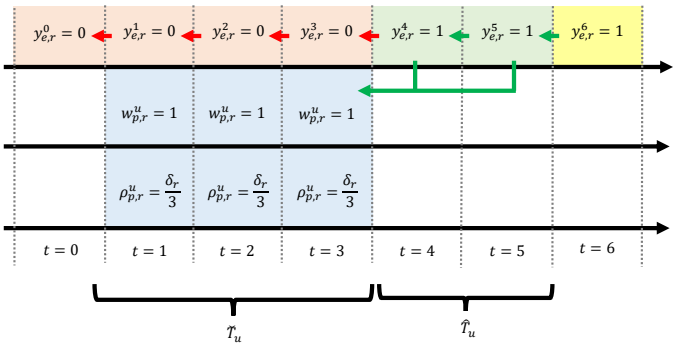


Fig. 2: Illustration of layer download formulation.

and continues with a constant rate until $\check{\tau}_u - 1$. The constant rate, denoted by ρ^u_r, is computed as follows to ensure that the layer r becomes fully downloaded and available in time slot $\check{\tau}_u$ when chain u starts its operation:

$$\rho^u_r = \frac{\delta_r}{|\check{\mathcal{T}}_u|}. \tag{8}$$

Consequently, $w^u_{p,r} = 1$ ensures that layer r is available for the usage of chain u (and possibly other chains) during time slots from $\check{\tau}_u$ to $\hat{\tau}_u$. After time slot $\hat{\tau}_u$, the layer remains in edge server e if another chain is scheduled to use it or freeing storage is not necessary. Later, for computing the bandwidth consumption and enforcing the bandwidth constraints, we consider a traffic with rate ρ^u_r on all links in path p during time slots from $\check{\tau}_u$ to $\hat{\tau}_u$. To ensure that a path is selected for downloading a required but missing layer, we use the following constraint:

$$y^t_{e,r} \le y^{t-1}_{e,r} + \sum_{u\in\mathcal{U}: t\in\hat{\mathcal{T}}_u} \sum_{p\in\mathcal{P}^\Theta_e} w^u_{p,r}, \quad r\in\mathcal{R}, e\in\mathcal{E}, t\in\mathcal{T} \tag{9}$$

This constraint states that if layer r is not available in time slot $t-1$, but the layer is required by one or more chains in time slot t (*i.e.*, $t \in \hat{\mathcal{T}}_u$), one of those chains should initiate the download by setting $w^u_{p,r}$ to one.

Example. Please consider Fig. 2 which illustrates the interaction of decision variables that control the availability of a layer in an edge server. In time slot zero, layer r is not available in edge server e. Note that the layer is also unavailable in time slots 1, 2, and 3. In time slot 1, a request for service chain u arrives to the network. This chain starts its operation in time slot 4 and terminates in time slot 5. If u requires layer r in edge server e, a path p from e to the cloud should be established to download the required layer (*i.e.*, set $w^u_{p,r}$ to one for an appropriate path p). This will allocate $\rho^u_r = \frac{\delta_r}{3}$ units of bandwidth on path p from time slot 1 to 3 for downloading layer r. Since the sum $\sum_{u\in\mathcal{U}:t\in\hat{\mathcal{T}}_u} \sum_{p\in\mathcal{P}^\Theta_e} w^u_{p,r}$ is equal to one for $t=4$ and $t=5$ (as $4,5\in\hat{\mathcal{T}}_u$), the values of $y^4_{e,r}$ and $y^5_{e,r}$ are set to one, which indicates the availability of layer r in server e in those time slots. Note that the layer can remain available in time slot 6 if it is not necessary to delete it and download another layer. We should emphasize that the availability of the layer in time slot 6 is due to its availability in the previous time slot 5. □

We also enforce each layer to be downloaded over at most one path:

$$\sum_{p\in\mathcal{P}^\Theta_e} w^u_{p,r} \le 1, \quad u\in\mathcal{U}, e\in\mathcal{E}, r\in\mathcal{R}, \tag{10}$$

and to respect the storage capacity of edge servers, we write the following constraint:

$$\sum_{r\in\mathcal{R}} y^t_{e,r}\delta_r \le D_e, \qquad e\in\mathcal{E}, t\in\mathcal{T}. \tag{11}$$

C. Traffic Routing

We should route incoming traffic of each chain through all of its VNFs in the specified order. To this end, the traffic from the entry point O_u first should reach where the first VNF is being placed. Then, the output traffic of VNF i should enter VNF $(i+1)$. Finally, the output traffic of the last VNF should go back to entry point O_u. Thus, we define binary decision variable $q^i_{u,p}$ to show whether or not the traffic towards the i-th VNF of chain u uses path $p\in\mathcal{P}$. Also, we let $q^{S_u+1}_{u,p}$ show whether or not the traffic from the last VNF uses path $p\in\mathcal{P}$ to go back to entry point O_u. This definition unifies the notation of routing toward VNFs and the entry point. The following constraints ensure that traffic routing satisfies the described requirements.

$$\sum_{p\in\mathcal{P}^n_m} q^1_{u,p} = v^1_{u,n}, \qquad m=O_u, u\in\mathcal{U}, n\in\mathcal{N} \tag{12}$$

$$\sum_{p\in\mathcal{P}^m_n} q^{S_u+1}_{u,p} = v^{S_u}_{u,n}, \qquad m=O_u, u\in\mathcal{U}, n\in\mathcal{N} \tag{13}$$

$$v^i_{u,n} + v^{i+1}_{u,m} - 1 \le \sum_{p\in\mathcal{P}^m_n} q^{i+1}_{u,p}, \quad u\in\mathcal{U}, i\in[S_u-1], n\ne m\in\mathcal{N} \tag{14}$$

Specifically, constraint (12) ensures that a path exists from the entry point to the location of the first VNF. Constraint (13) ensures the existence of a path from the location of the last VNF to the entry point O_u. Constraint (14) ensures that there is a path between consecutive VNFs if they are not co-located.

To guarantee the bandwidth constraints, first we compute the rate of traffic that enters each VNF as follows:

$$\lambda^i_u = \lambda_u \prod_{j\in[i-1]} \beta^j_u, \tag{15}$$

Then, we can use the following constraint to ensure that in each time slot the capacity of each link in the network is respected:

$$\sum_{u\in\mathcal{U}:t\in\check{\mathcal{T}}_u} \sum_{r\in\mathcal{R}} \sum_{p\in\mathcal{P}:\ell\in\mathcal{L}_p} w^u_{p,r}\rho^u_r + \sum_{u\in\mathcal{U}:t\in\hat{\mathcal{T}}_u} \sum_{p\in\mathcal{P}:\ell\in\mathcal{L}_p} \sum_{i\in[S_u]} q^i_{u,p}\lambda^i_u \le b_\ell, \quad \ell\in\mathcal{L}, t\in\mathcal{T} \tag{16}$$

Notice that for each time slot t the first term in the left-hand side of (16) only considers downloading the required layers for those chains that have arrived to the network but have not started their operation yet (*i.e.*, $t\in\check{\mathcal{T}}_u$). Thus, this

term computes the bandwidth consumption due to download of layers in time slot t. On the other hand, the second term only considers those chains that are operational in the time slot (*i.e.*, $t \in \hat{\mathcal{T}}_u$). Thus, this term computes the bandwidth consumption due to traffic that passes between subsequent VNFs of operational chains in time slot t. The sum of these two terms on each link should be less than the capacity of that link in all time slots.

D. Delay

To guarantee the end-to-end propagation delay, we should sum the propagation delay of all links in the paths selected for chaining the VNFs as follows:

$$\sum_{i\in[S_u]}\sum_{p\in\mathcal{P}} q^i_{u,p}\sum_{\ell\in\mathcal{L}_p} d_\ell \le \Phi_u, \qquad t\in\mathcal{T}, u\in\mathcal{U} \tag{17}$$

Note that it is straightforward to use the modified Amdahl's law, similar to [36], and extend the delay equation to include the transmission and processing delays. However, they do not change the complexity of the problem and for simplicity we omit them here.

E. Objective

Since each chain specifies its CPU and memory requirement, the total computational and memory consumption are fixed and independent of VNF placement. Therefore, we consider the total bandwidth consumption as the objective and minimize it as follows:

$$\text{(CCO):} \qquad \text{Min.} \sum_{p,i} q^i_{u,p}|\mathcal{L}_p|\lambda^i_u \tag{18}$$

Theorem 1. *CCO is NP-hard.*

Proof. We proof the NP-hardness by a reduction from the well-known Usplittable Flow Problem (UFP) [37]. UFP considers a network of nodes and links and a set of pair of nodes where each pair has a demand and each link has a capacity. We can consider each pair to be a chain. Source node of the pair is represented by the entry point O_u and demand of the pair is represented by the chain input traffic λ_u. Each chain has only one VNF with $\beta^i_u = 0$, so there is no returning traffic from the VNF to the entry point. The destination of each pair is modelled by an edge server. To ensure that the VNF of each chain is placed on the corresponding edge server, we assign distinct positive integer identifiers to chains and let CPU requirement of each chain's VNF be equal to the assigned identifier. Equip each edge server that is the destination node of a pair with a CPU capacity equal to chain's identifier. Thus, each VNF is forced to be placed on the destination node. If we could solve CCO in polynomial time, UFP is solvable in polynomial time, which is a contradiction. □

V. Proposed Algorithm

We mathematically formulated the containerized chain orchestration problem in the previous section and proved its NP-hardness in Theorem 1. Thus, we know that it not possible to find its exact solution in reasonable time for arbitrary inputs. Therefore, we present a time-efficient and low-complexity solution in this section to solve CCO. To this end, we take into account that in most real-world systems chains arrive to the network sequentially and information about future chains is not available a priori. Motivated by this fact, we design a solution that processes one chain at a time. Also, we use the framework of linear programming (LP) relaxation and rounding [38] to design an efficient solution method.

In the following, we first make an overview of the proposed algorithm in Subsection V-A. Then, we explain three important aspects of this algorithm, *i.e.*, `Round` *Subroutine, Scaling Link Capacities, and Layer Eviction* in more detail in Subsections V-B to V-D.

A. Algorithm Overview

General Sketch. The high-level steps of our proposed algorithm is outlined in Algorithm 1. The general sketch of the algorithm is as follows. First we construct the integer program formulation for the chain (line 1) in hand but discard the integrality constraints and allow decision variables to take fractional values between zero and one (line 2). Then, we solve the obtained linear program with an off-the-shelf optimization solver, which is an efficient procedure. Note that linear programs are solvable in polynomial time. Then, we perform a rounding procedure to round the values of decision variables to either zero or one through an iterative low-complexity procedure while ensuring the feasibility of problem constraints.

Sequential VNF Placement. In each iteration of our rounding procedure, we focus on one VNF along the chain and attempt to fix the location of the VNF, select appropriate paths to download the required container layers, and select a path from the previous VNF in the chain, or the entry point in the case that the VNF is the first one in the chain. Variable i defined in line 3 of Algorithm 1 shows the considered VNF in each iteration. We start from the first VNF and proceed one by one along the chain. Subroutine `round` that is invoked in line 6 handles the placement, layer download, and routing for the considered VNF in each iteration. The procedure gets as the input a value $\varepsilon < 1$ that is used to control the failure probability of the `round` subroutine. The detailed description of this subroutine is given in the subsection V-B.

Backtracking Procedure. Subroutine `round` returns a status to show whether or not it was able to perform all the required allocations necessary for successful placement of the specified VNF. If the status indicates a successful operation, we proceed to the next iteration to handle the next VNF. However, if `round` fails, we perform a limited backtrack by reverting the decisions made in the most recent Γ iterations. Γ should be fine tuned in practice. Our experiment in the Section VI shows that $\Gamma = 1$ or 2 is effective and beneficial. After performing a backtrack, we do not allow another backtrack before completing $\Gamma + 1$ iterations to ensure efficiency. Therefore, if `round` fails and a backtrack is not allowed, our algorithm fails. This means that we should reject the chain request. In this case, the chain can either reduce its resource demands and make a new request immediately or try in a later time when the network is less busy.

Algorithm 1: RCCO: Rounding-based Containerized Chain Orchestration

Input : $\mathcal{N}, \mathcal{L}, \mathcal{P}$,u, ε, Γ
Output: Status of allocation of $u \in \mathcal{U}$

M $\leftarrow$ `ilp_model`(u) /* Defined in Section IV */
`relax`(M)
$i \leftarrow 1$
$g \leftarrow \Gamma$
while $i \leq S_u$ **do**
  status $\leftarrow$ `round`(u, i, M, ε)
  if status = `FEASIBLE` **then**
    $i \leftarrow i + 1$
    $g \leftarrow \min\{g + 1, \Gamma\}$
  end
  else if $i > 1$ *and* $g = \Gamma$ **then**
    $g \leftarrow \max\{-1, g - i - 1\}$
    $j \leftarrow \max\{i - \Gamma, 1\}$
    Unfix all variables for VNFs j to i
    Disallow the placement of VNF j on its currently chosen location
    $i \leftarrow j$
  end
  else
    return FAIL
  end
end
Allocate resources to u based on M
return SUCCESS

B. `Round` Subroutine

Our proposed scheme named RCCO, presented in Algorithm 1, manages VNFs in a chain one by one beginning from the first VNF. In this subsection, we explain the `round` subroutine in more detail. Note that RCCO passes the linear program model **M** to `round` in a way that modifications done by `round` (*e.g.*, fixing a decision variable) also affect the model in RCCO. `round` is responsible for placing the considered VNF in the i^{th} iteration of the algorithm, connecting it to the previous VNF in the chain, and selecting download paths for missing layers of the VNF. In `round`, we fix the location of the i-th VNF in the chain and the required download paths to acquire missing layers. Then, we connect VNF i to VNF $(i-1)$ that was assigned to a node in the previous iteration. The first and last VNFs are connected to the entry point O_u. At start, `round` re-solves the LP model to re-adjust the value of remaining unfixed decision variables. Since the changes of the linear program are small, the required time to re-compute the value of unfixed decision variables is negligible compared to the time required for solving the linear program for the first time. Then, `round` selects appropriate decision variables based on their fractional values and fixes them to ensure the placement of the VNFs. In this subsection, we present the fractional value of decision variables by putting a tilde on top of them. For example $\widetilde{v}^i_{u,n}$ shows the fractional value of decision variable $v^i_{u,n}$ obtained from solving the linear program.

The fractional value of decision variables are indicators of suitability of network entities and paths. Thus, it is reasonable to use the decision variables with the maximum value to handle the placement of VNF i. However, this choice can lead to the violation of capacity constraints. To avoid the constraint violation, at the beginning of the i-th iteration, we eliminate the infeasible options related to the placement of i-th VNF before solving the linear program.

Algorithm 2: `round`: Rounding Procedure

Input : Chain u, VNF Id i, LP model M, ε
Output: Status of rounding M for placing VNF i

Eliminate all infeasible edge servers
Eliminate all infeasible chaining paths
$\gamma \leftarrow$ based on ε and (33)
Scale the capacity of all links during $\check{\mathcal{T}}_u$ by $(1-\gamma)$
$\{\widetilde{v}^i_{u,n}, \widetilde{w}^u_{p,r}, \widetilde{q}^i_{u,p}\} \leftarrow$ `solve-lp`(M)
if M is `INFEASIBLE` **then**
  return `S-Failure`
end
$n_i \leftarrow \text{argmax}_{n' \in \mathcal{N}}\{\widetilde{v}^i_{u,n'}\}$
$p_i \leftarrow \text{argmax}_{p' \in \mathcal{P}^{n_i}_{n_{i-1}}}\{\widetilde{q}^i_{u,p'}\}$
for $r \in Q^i_{u,n_i}$ **do**
  Fix $w^u_{p,r} = 1$, with probability $\widetilde{w}^u_{p,r}$
  if *Constraint* (16) *is violated* **then**
    Revert all changes
    return `R-Failure`
  end
end
Fix v^i_{u,n_i} to 1
Fix q^i_{u,p_i} to 1
return `FEASIBLE`

Edge Servers. To eliminate the infeasible edge servers, let $\check{P}_e$, $\check{A}_e$, and $\check{D}_e$ show the available CPU, RAM, and disk capacity of edge server e during time slots in $\hat{\mathcal{T}}_u$. Also, let $Q^i_{u,e}$ show the missing layers that are required for placing i-th VNF of chain u on edge server e. Then, we eliminate the possibility of placing VNF i on edge server e by enforcing the constraint $v^e_{u,i} = 0$ for servers that their CPU is not sufficient, *i.e.*, $\pi^i_u > \check{P}_e$, their RAM is not sufficient, *i.e.*, $\alpha^i_u > \check{A}_e$, or their disk is not sufficient, *i.e.*, $\sum_{r \in Q^i_{u,e}} \delta_r > \check{D}_e$.

Chain Paths. We eliminate the possibility of selecting a path for connecting VNFs, if its capacity is not sufficient. To this aim, let $\check{b}_\ell$ and $\hat{b}_\ell$ be the available capacity of link ℓ during time slots in $\check{\mathcal{T}}_u$ and $\hat{\mathcal{T}}_u$, respectively. Then, for path p, we set $q^i_{u,p} = 0$ if it has a link ℓ for which $\hat{b}_\ell < \lambda^i_u$.

Download Paths. Improving the probability of successful rounding of download variables, $w^u_{p,r}$, is more complicated because missing layers can be downloaded over multiple paths and a path can be used to download multiple layers. We can eliminate download paths that can not be used to download even the smallest missing layers, however, it is not helpful. Thus, we adopt another strategy. We scale the capacity of all links during time-slots in $\check{\mathcal{T}}_u$ by a multiplier $(1-\gamma) \leq 1$ that its computation is presented in Subsection V-C. Scaling link capacities before solving the linear program helps to obtain fractional values that respect link capacities after rounding them to one.

After eliminating infeasible edge servers and chain paths, and scaling links in the download paths, we solve the linear program model. At this stage we get fractional solutions that have higher chance of being rounded without violating a constraint. If the model becomes infeasible at this step, even a fractional solution is not possible and thus we terminate the routine and return the flag `S-Failure` to show a Solve failure, which triggers a backtrack in 1. However, if the model is feasible, we get fractional values of decision variables. Based on them, we select the candidate placement location, and chaining path as follows:

$$n_i \leftarrow \underset{n' \in \mathcal{N}}{\operatorname{argmax}} \{\widetilde{v}^i_{u,n'}\}. \tag{19}$$

$$p_i \leftarrow \underset{p' \in \mathcal{P}^{n_i}_{n_{i-1}}}{\operatorname{argmax}} \{\widetilde{q}^i_{u,p'}\}. \tag{20}$$

Then, we use the fractional values of $w^u_{p,r}$ as probabilities and select paths based on these probabilities.

$$w^u_{p,r} = 1, \qquad \text{with probability } \widetilde{w}^u_{p,r} \tag{21}$$

After selecting the paths, we examine bandwidth constraints (16) with original values of bandwidth before scaling them down. If constrains are not satisfied, we revert changes applied to fix the download paths and return the `R-Failure` flag to indicate a failure of Rounding, which triggers a backtrack in Algorithm 1. If constraints are satisfied, we fix the location for the i-th VNF, *i.e.*, n_i, by adding the corresponding constraint:

$$v^{n_i}_{u,i} = 1. \tag{22}$$

Then, we fix the chaining path with:

$$q^{p_i}_{u,i} = 1. \tag{23}$$

C. Scaling Link Capacities

Consider that Q^i_{u,n_i} is the set of missing container layers needed for placing i-th VNF of chain u on edge server n_i. As the constraint (9) in our linear program selects fractional paths for all missing layers (if it is feasible) and considering the constraint (10), we may write the following equation for each missing layer:

$$\sum_{p \in \mathcal{P}^{\Theta}_{n_i}} \widetilde{w}^u_{p,r} = 1, \qquad r \in Q^i_{u,n_i}. \tag{24}$$

The fractional values of $\widetilde{w}^u_{p,r}$ can be considered as a discrete random variable and thus the probability of selecting a download path for a missing layer may be calculated from its probability mass function. Then, we define random variable Z^r_ℓ as follows to specify the load of downloading missing layer r on link ℓ in the network:

$$Z^r_\ell = \begin{cases} \rho^u_r, & w.p. \frac{\sum_{p \in \mathcal{P}^{\Theta}_{n_i}: \ell \in \mathcal{L}_p} \widetilde{w}^u_{p,r}}{\sum_{p \in \mathcal{P}^{\Theta}_{n_i}} \widetilde{w}^u_{p,r}} \\ 0, & o.w. \end{cases} \tag{25}$$

Notice that $Z^r_\ell = \rho^u_r$ indicates that missing layer r is downloaded with rate ρ^u_r over a path that contains link ℓ and otherwise Z^r_ℓ is equal to 0. Downloading the missing layers based on the probabilistic strategy of RCCO, burdens a load on each link that is equal to the sum of variables Z^r_ℓ for each link (*i.e.*, $\sum_{r \in Q^i_{u,n_i}} Z^r_\ell$). Before solving the linear program, we scale down the link capacities by multiplying them with $(1-\gamma)$ and thus we need to modify the link capacity constraint presented in (16) as below:

$$\sum_{r \in Q^i_{u,n_i}} \sum_{p \in \mathcal{P}^{\Theta}_{n_i}: \ell \in \mathcal{L}_p} \widetilde{w}^u_{p,r} \rho^u_{p,r} \le (1-\gamma)\check{b}_\ell. \qquad \ell \in \mathcal{L} \tag{26}$$

From (24) and (26) we can write the following relation about the expectation of the sum of random variables Z^r_ℓ for each link ℓ:

$$\mathrm{E}\Big[\sum_{r \in Q^i_{u,n_i}} Z^r_\ell\Big] \le (1-\gamma)\check{b}_\ell. \tag{27}$$

The probability of exceeding the capacity is greater when the expectation is bigger. Thus, we focus on equality case: $\mathrm{E}\Big[\sum_{r \in Q^i_{u,n_i}} Z^r_\ell\Big] = (1-\gamma)\check{b}_\ell$. We employ the Hoeffding's inequality [39] to characterize the probability of exceeding the capacity of link ℓ. To this end, we define the following values for each link:

$$n_\ell = |Q^i_{u,n_i}|, \tag{28}$$

$$\overline{\rho}_\ell = \max\{\rho^u_{p,r} | r \in Q^i_{u,e}, p \in \mathcal{P}^{\Theta}_e, \ell \in \mathcal{L}_p\}, \tag{29}$$

$$1 + \delta = \frac{1}{(1-\gamma)}, \tag{30}$$

where, n_ℓ is the number of missing layers, $\overline{\rho}_\ell$ is the maximum rate of download on link ℓ due to any missing layer, and $1+\delta$ is a multiplier that reverses the effect of scaling down the link capacity employed to obtain the probability of exceeding the actual link capacity. Then, we can write the Hoeffding's inequality for the sum of variables Z^r_ℓ for link ℓ as follows:

$$\Pr\Big\{\sum_{r \in Q^i_{u,n_i}} Z^r_\ell \ge \check{b}_\ell\Big\} < \exp\{-\frac{2\gamma^2\check{b}^2_\ell}{n_\ell \overline{\rho}^2_\ell}\}. \tag{31}$$

To obtain γ for a given ε we write:

$$\exp\{-\frac{2\gamma^2\check{b}^2_\ell}{n_\ell \overline{\rho}^2_\ell}\} \le \varepsilon, \tag{32}$$

which gives:

$$\gamma \ge \sqrt{-\frac{n_\ell \overline{\rho}^2_\ell \ln \varepsilon}{2\check{b}^2_\ell}}. \tag{33}$$

D. Layer Eviction

Once the service of a chain terminates, it will release the allocated resources. However, its downloaded container layers will remain on the edge server. RCCO keeps the downloaded layers to avoid downloading them again in the future. However, if other chains arrive to the network and require different layers, RCCO may need to delete existing layers to free space and download those layers. Instead of deleting layers at random, RCCO employs a lightweight mechanism to compute a popularity for layers that have been downloaded to edge servers so far. RCCO keeps this information in each edge server individually and in a distributed manner. RCCO

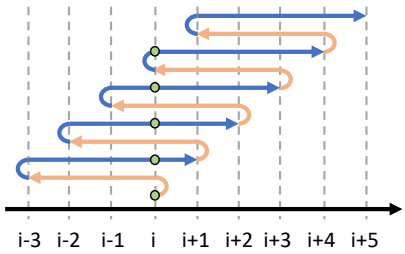


Fig. 3: Limited backtrack with $\Gamma = 3$ causes the placement of i-th VNF to be repeated at most 5 times ($\Gamma + 2$).

creates a table in each edge server and each time a new layer is downloaded, RCCO writes an ID that uniquely identifies the layer in the table. Associated with layer ID RCCO keeps a popularity metric of that layer which is initialized to zero. Then, each time that a layer is used to instantiate a VNF on the edge server, RCCO increments the popularity metric in the table by:

$$\text{Popularity} \leftarrow \epsilon \times \text{Popularity} + 1,$$

where, "Popularity" is the current value and ϵ is a coefficient less than one. We note that this equation is inspired by the cumulative discounted reward that is employed in the theory of reinforcement learning. When a chain arrives and we solve the optimization model, the location of each VNF and required layers become specified. For each edge server, if it can host specified VNFs without deleting a layer, we keep all unused layers. However, if it becomes necessary to delete some layers to download new layers, we start by the least popular layer and delete more layers in that order until the required space becomes available.

Theorem 2. *RCCO runs in $O(S_u \Gamma T_{lp})$, where T_{lp} is the time complexity of solving relaxed CCO.*

Proof. We have to place S_u VNFs to handle chain u. Fig. 3 shows that the placement of each VNF can at most repeat $\Gamma+2$ times due to limited backtrack mechanism. To place each VNF, we call `round` where it eliminates infeasible servers (at most $O(|\mathcal{N}|)$), infeasible chaining paths from previous location (at most $|\mathcal{P}|$) and scales link capacities (*i.e.*, $O(|\mathcal{L}|)$). Then, `round` solves the linear program. After that, for each missing layer a path is selected (there are at most $\max_{u,i}\{|\mathcal{R}_u^i|\}$ missing layers). After selecting a path for a missing layer, we should check the link capacities along that path to detect capacity violation. There are at most $O(|\mathcal{L}|)$ links in any path. The time complexity of solving a linear program with interior-point methods is $O(n^{3.5})$, where n is the number of decision variables. Since the complexity of solving the linear program is higher than any of pre-processing and post-processing steps, the complexity of RCCO is $O(S_u \Gamma T_{lp})$. □

VI. Performance Evaluation

In this section, we present the performance evaluation of RCCO which was introduced in Section V. We implemented all algorithms in Python 3.7 and used Gurobi 9.0 [40] as the solver of ILPs and LPs. All the instructions and codes required for reproducing the presented results are available in [41]. Computations are carried out on a computer with an Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90 GHz, 16 GB of RAM, and 64-bit Windows Server 2016 operating system.

TABLE IV: Important Parameters

Parameter	Value	Parameter	Value
No. VNFs	50	b_ℓ	[0.1, 1] Gbps
VNF RAM	[0.25, 0.5] GB	$\check{\tau}_u$	τ_u + [5, 8]
No. Layer/VNF	[5, 12]	$\hat{\tau}_u$	τ_u + [8, 21]
No. Layers	200	Layer Size	[50, 350] Mb
SFC length	[2, 7]	Φ_u	[500, 2000] ms
β_u^i	[0.8, 1.05]	λ_u	[1, 5] Mbps
VNF CPU	[0.25, 0.5] GHz	Server CPU	[16, 32] GHz
Server RAM	[4, 32] GB	Server Disk	[20, 30] Gb
Inter-arrival	exp (1)		

A. Evaluation Setup

We observed that container layers are different in terms of applicability and usage. For example, operating system layers such as CentOS are more commonly used in the real-world container images. To show this diversity, we employ a popularity-based approach to construct the VNFs from layers. To this end, we randomly assign a natural number $n_r \in \mathbb{N}$ to each layer and then let the probability of using layer n_r in a VNF be proportional to $1/n_r$. This is the famous Zipf's law [42] which is widely used for modeling popularity in physical and social sciences. We assume that the number of service request arrivals in time follows a Poisson random process. Thus, we use exponential distribution to generate requests' inter-arrival times and thus τ_u's to achieve the designated arrival rate. Important parameters are presented in Table IV.

B. Implemented Schemes

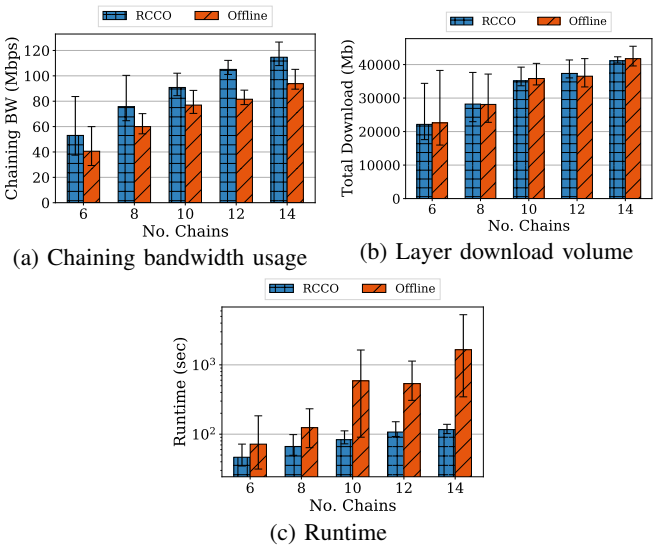
We compare RCCO with the following algorithms to demonstrate its effectiveness:

- **Offline**: This algorithm assumes that the information about all chains are available a priori and solves the optimization problem described in Section IV with the solver.
- **Online**: This algorithm solves the optimization problem described in Section IV with the solver for each chain upon arrival and then fixes all the corresponding decision variables. This algorithm is the natural extension of the above Offline algorithm to online setting.
- **V**NF **P**acker (**VP**): This algorithm packs subsequent VNFs of a Chain in the same edge server if possible. Otherwise, it selects the next edge server that has sufficient capacity for placing the remaining VNFs.
- **I**terative **G**reedy **A**lgorithm (**IGA**): This algorithm places VNFs on edge servers based on the greedy metric described in [7].
- **NoShare**: This algorithm follows the same procedure as RCCO, but does not allow layer sharing among VNFs.

C. Metrics

We consider the following metrics in our comparisons. We report averages and 95% confidence intervals of these metrics obtained from experiments.

- **Accept rate**: The ratio between the number of chains that are accepted to all chains that arrive to the network.



(a) Chaining bandwidth usage (b) Layer download volume

(c) Runtime

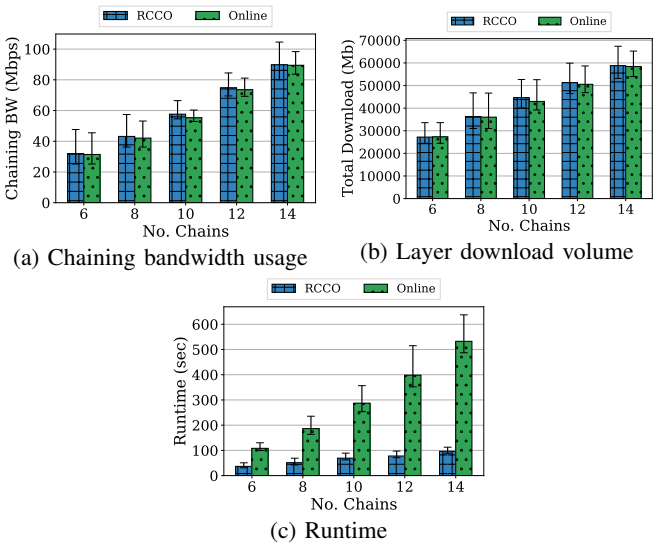
Fig. 4: Performance comparison of RCCO and Offline algorithm. Note that the y-axis of 4(c) displays a logarithmic scale.

- **Chain bandwidth**: The amount of bandwidth used to satisfy the bandwidth demand of chains. In some cases that two consecutive VNFs are placed inside the same server, no bandwidth is consumed. However, in some cases the two VNFs are connected via a path that has more than one link. This metric is minimized in the objective function of the problem in equation (18).
- **Download volume**: Total amount of container layer data downloaded from cloud to edge servers to instantiate VNFs.
- **Runtime**: Total time spent for processing all chains.

D. Performance Accuracy

In this subsection, we compare RCCO with offline and online algorithms derived from solving an integer linear program exactly. Since these algorithms solve the problem optimally, they serve as credible benchmarks to demonstrate the accuracy of RCCO. As exact solvers are not scalable, we have to restrict the comparisons to small scale problem instances.

Offline. In this part, we compare RCCO with Offline that is based on the solver and assumes the knowledge of all chains a priori. Since Offline is not scalable, we test with 6 to 14 number of chains. Also, as the main objective is minimizing the chain bandwidth, we first give all chains to RCCO and determine the chains that it accepts. Then, we give the accepted chains to Offline. This step ensures that RCCO and Offline accept the same set of chains and allows us to compare their ability of efficiently placing VNFs and minimizing the chain bandwidth. Fig. 4 shows the result of comparisons. Fig. 4(a) indicates that for the same number of chains, Offline at most reduces chain bandwidth by about 23% and at least by 15% compared to RCCO. The average bandwidth reduction by Offline compared to RCCO is about 20%. We emphasize that the Offline approach is not realistic as chains arrive sequentially and knowledge about future chains is not available in practice. Fig. 4(b) shows that the layer download volume is similar for both approaches. Finally, we can see in Fig. 4(c) that the runtime of Offline increases significantly with the number of chains. As the number of chains increases by 2.3x, the runtime of RCCO also increases by 2.5x, which shows a preferable linear behavior. However, the runtime of Offline increases by 23x. The runtime of RCCO is 35% smaller than the runtime of Offline for handling 6 chains while the difference becomes 92% for 14 chains. Furthermore, we observe that although the average runtime of Offline for handling 14 chains is about 1600 seconds, its 95% confidence interval includes 5000 seconds. This observation shows that for some instances Offline's runtime is significantly long. Despite the 20% bandwidth reduction of Offline compared to RCCO, the significant better runtime performance of RCCO and non-realistic knowledge about future chains employed in Offline demonstrate the effectiveness and scalability of RCCO.



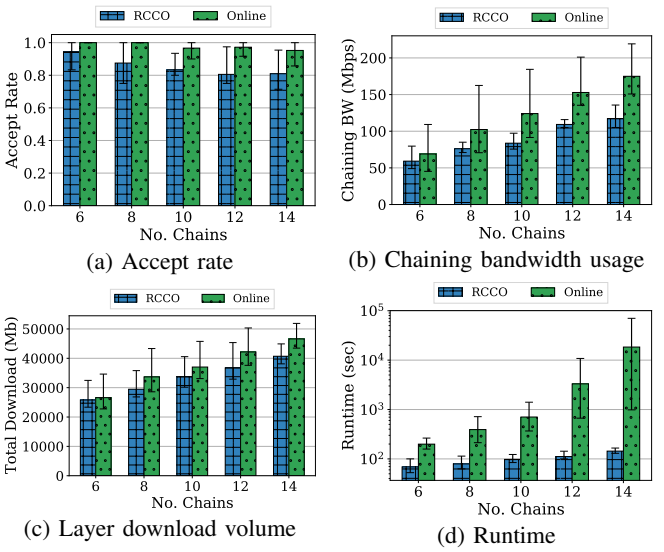
(a) Chaining bandwidth usage (b) Layer download volume

(c) Runtime

Fig. 5: Performance comparison of RCCO and Online algorithm in the first setting with bandwidths of links large enough to allow both RCCO and Online accept all incoming chains.

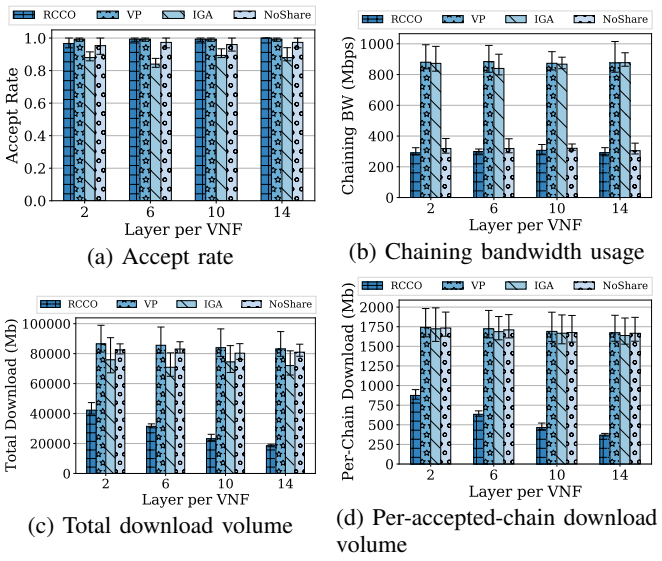
Online. We compare RCCO and Online in two different settings. In the first one, we set the bandwidths of links large enough to allow both RCCO and Online accept all incoming chains. This setting allows us to compare RCCO and Online in terms of their ability to efficiently allocate physical resources. In the second setting, we use values for the bandwidths of links from the interval $[0.1, 1]$ Gbps as indicated in Table IV, and consequently due to insufficient resource, some chains will be rejected. Since Online uses the solver to directly solve the optimization program for each chain, it will uses resources more efficiently and accepts more chains. The second setting allows us to compare RCCO with the best obtainable result in a situation closer to real-world.

Fig. 5 shows the result of comparisons in the first setting. Fig. 5(a) indicates that for the same number of chains, RCCO uses slightly more bandwidth which in the worst-case is about 4% and on average is about 2%. Same result is observable in Fig. 5(b) that compares the download volumes. The result of these figures shows that RCCO is very close to the optimal solution of the problem in the online setting. Fig. 5(c) compares the runtime of RCCO and Online in the first setting. We observe that the runtime of RCCO exhibits a linear increase and increases by 2.6x as the number of chains increases by 2.3x.



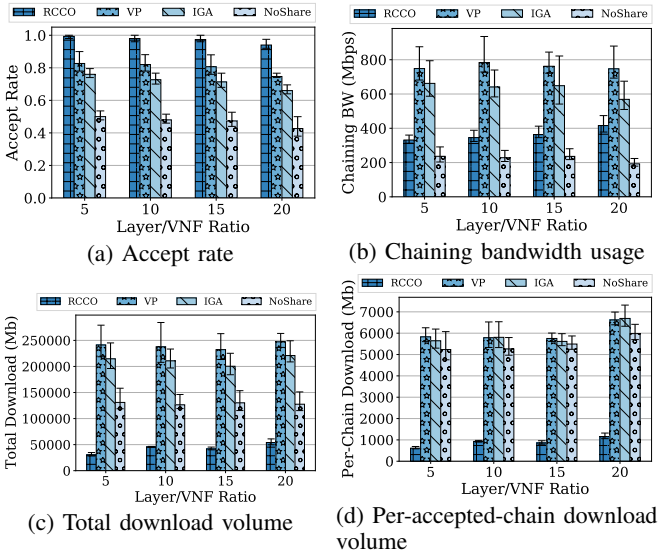
(a) Accept rate (b) Chaining bandwidth usage (c) Layer download volume (d) Runtime

Fig. 6: Performance comparison of RCCO and Online algorithm in the second setting with limited bandwidths for links generated from the interval $[0.1, 1]$ Gbps. Note that the y-axis of 6(d) displays a logarithmic scale.



(a) Accept rate (b) Chaining bandwidth usage (c) Total download volume (d) Per-accepted-chain download volume

Fig. 7: Effect of splitting VNFs to more layers. As VNFs are split into more layers, the reuse and sharing opportunity increases, which leads to total and per-VNF layer download.



(a) Accept rate (b) Chaining bandwidth usage (c) Total download volume (d) Per-accepted-chain download volume

Fig. 8: Effect of increasing the total number of layers compared with the total number of VNFs.

However, the runtime of Online increases by 4.9x. Online's runtime on average is 327% higher than RCCO in this setting.

Fig. 6 shows the results of comparison with Online in the second setting. We see in Fig. 6(a) that Online algorithm accepts on average about 12% more chains. As the result of higher accept rate, Online consumes 37% more bandwidth to chain the VNFs and downloads 11% more layer data which are demonstrated in Figs. 6(b) and 6(c). Fig. 6(d) shows the runtime comparison of RCCO and Online in the second setting. Since the bandwidths of links are limited, finding a solution is more challening compared with the first setting. We observe that the runtime of Online for handling 14 chains is close to 20000 seconds and the 95% confidence interval includes 70000 seconds.

E. Performance Comparison

In this subsection, we compare RCCO with three heuristic algorithms introduced in Subsection VI-B that are scalable and run in polynomial time. Thus, these algorithms have no scalability concern and are applicable in practice.

Effect of the Number of Layers Per VNF. In this experiment, we fix the size of VNFs to 420 Mb and split a VNF into 2, 6, 10 and 14 layers. We expect to observe a reduction in the download volume as number of layers per VNF increases and thus the opportunity of reusing layers and sharing them among VNFs increases. First we evaluate the accept rate and chain bandwidth consumption. Fig. 7(a) shows that RCCO and VP achieve higher accept rate compared to IGA and NoShare. Fig. 7(b) shows that RCCO and NoShare reduce the chain bandwidth significantly compared to VP and IGA as they consider the chain bandwidth consumption in their algorithms. Figs. 7(c) and 7(d) show that RCCO downloads less than half of other algorithms in general and per-chain. Also, these figures show that as the number of layers increases, the download volume for RCCO is reduced, while other algorithms can not exploit the layering benefits properly. We can observe that RCCO visibly outperforms other algorithms when we consider different aspects of accept rate, chain bandwidth and download volume.

Effect of Total Number of Layers. In this experiment, we examine the effect of increasing total number of layers. We fix the total possible number of VNFs that form service chains to 10 and consider scenarios with 50, 100, 150 and 200 total number of possible container layers in the whole network. As the number of layers increases, the sharing between VNFs decreases and we expect to observe a gradual increase in the download volume. Notice that we build the VNFs from the pool of available layers according to the Zipf's law. Therefore, when the total number of layers increases the number of unique layers in VNFs also increases which reduces the

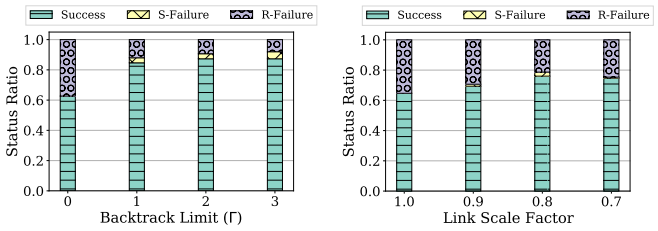
opportunity of layer sharing. Fig. 8(a) shows that as the total number of layers increases the accept rate experiences a slight decrease. Nevertheless, RCCO outperforms other algorithms by at least about 17% on average. Interestingly, despite higher accept rate, RCCO also consumes significantly less bandwidth to route traffic between VNFs and download the required layers. Fig. 8(b) shows that the bandwidth consumption for chaining VNFs increases by about 24% as the number of layers quadruples. On the other hand, the bandwidth consumption of other algorithms slightly decreases which is due to their lower accept rate. Nonetheless, RCCO consumes about 60% less bandwidth in comparison while providing better accept rate. A similar trend is observable in Fig. 8(c). As the total number of layers increases compared with the number of VNFs, the layer sharing becomes more difficult and we can see an increase in the amount of container layer data. Specifically, we observe that the download volumes increases by about 89%, as the total number of layers increases from 50 to 200. The download volume of other algorithms do not change significantly which implies that they do not appropriately exploit the container layer sharing especially as we see that they download at least 4.3x more container data compared with RCCO.

F. Evaluation of RCCO components

In what follows, we investigate the effect of several important components of RCCO introduced in Section V on its performance.

Backtrack Limit. In this experiment, we explore the effectiveness of limited backtrack strategy (refer to Subsection V-A) employed in RCCO. To this end, we run RCCO with different backtrack limit values (i.e. different values of Γ). When Γ is zero, RCCO can not perform a backtrack. As we increase Γ, RCCO undoes more decisions but has to go further forward to be allowed to do another backtrack. Consequently, we expect to observe a turning point where too many backtracks becomes ineffective. To further explore the effect of backtracking, we define two different failures in RCCO as in Section V. S-Failure happens when RCCO can not solve the linear program after eliminating the infeasible options. R-Failure happens when RCCO can not round variables after solving the linear program. From Fig. 9(a) we can see that without backtracking, about 66% of chains are accepted successfully but remaining 34% failed due to R-Failure. As we allow backtracking, the accept rate increases by more than 20% compared to the algorithm that does not employ backtracking. We also observe about 3% S-Failures happen which means that even the linear program solver was not able to find a feasible solution and probably there was not sufficient amount of resources to accept the 3% of chains. In total, we can argue that backtracking has alleviated about 24% of R-Failures.

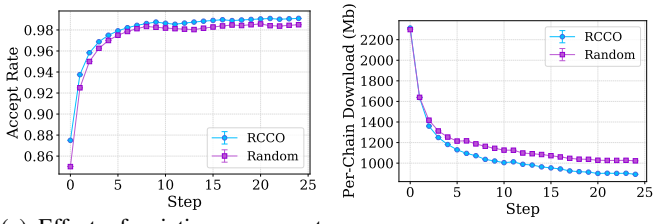
Link Scale Factor. In this experiment, we investigate the effect of the link capacity scale factor introduced in Subsection V-C and employed in RCCO to increase the success probability of the download path variable rounding. We set the backtrack limit to zero. Similar to our discussion of backtrack experiment, we record the frequency of S-Failure and R-Failure as link scale values increase. In Fig. 9(b), we observe that without



(a) Termination status for different backtrack limits

(b) Termination status for different link scale factors

Fig. 9: Effect of backtrack and scaling link capacities on the success of RCCO.



(a) Effect of eviction on accept rate

(b) Effect of eviction on layer download volume

Fig. 10: Random vs. popularity-based eviction strategy.

link scaling about 66% of chains are accepted successfully, while the remaining ones experience an R-Failure. However, after scaling the link capacities by 0.8, more than 78% of chains are accepted and 1.3% of them are rejected due to an S-Failure. Consequently, we can argue that link capacity scaling alleviated about 14% of R-Failures. These results confirm the effectiveness of the scaling strategy.

Layer Eviction Strategy. In this experiment, we examine the effect of the popularity-based layer eviction strategy that we introduced in Subsection V-D. In this regard, we record the accept rate and download volume during an interval that 1000 chains are handled. We report the results by averaging the metrics after handling every 40 chains. As the baseline, we re-use RCCO but instead of deleting the least popular layer we delete a layer at random. From Fig. 10(a) we can observe that both approaches admit relatively the same number of chains and attain the same accept rate. This is an expected result as deletion of unused layers has a limited influence on which chains can be accepted. Specifically, only when a chain arrives to the network and requires a container layer that is too large to be downloaded before the start of the chain's service and the layer is not available locally, the effect of deletion becomes evident on the accept rate. Nonetheless, RCCO based on popularity deletion consistently achieves a higher accept rate. However, eviction has a bigger effect on the amount of layer data that is downloaded from the cloud. Fig. 10(b) shows that RCCO can avoid downloading about 20% of layer data by keeping more popular layers instead of deleting unused layers at random.

VII. Conclusion

In this work, we considered the problem of orchestration of containerized VNFs in an edge-enabled network. We presented a novel ILP formulation to model the download and placement

of container layers that are required to instantiate virtualized network functions. Then, we designed an algorithm based on the linear program relaxation and rounding framework. To increase the efficiency of the method we equipped it with a limited backtrack capability and capacity scaling strategy. Then, we characterized several theoretical aspects related to performance of the proposed solution. We evaluated the proposed solution through simulating a typical real-world setting and demonstrated the ability of our solution to obtain considerable performance gains through container layer sharing and reuse. In our work, we also used a lightweight popularity strategy to delete unused layers that was a reasonable decision due to resource limitation at the edge. However, investigation of more sophisticated layer management strategies based on machine learning without putting too much stress on edge servers is an interesting future direction of our work.

REFERENCES

[1] Y. Liu, M. Peng, G. Shou, Y. Chen, and S. Chen, "Toward edge intelligence: multiaccess edge computing for 5G and internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6722–6747, 2020.

[2] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou, "A survey on edge computing systems and tools," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1537–1562, 2019.

[3] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[4] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *Proc. ACM SIGOPS/EuroSys*, 2007, pp. 275–287.

[5] Z. Tao, Q. Xia, Z. Hao, C. Li, L. Ma, S. Yi, and Q. Li, "A survey of virtual machine management in edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1482–1499, 2019.

[6] D. Mauro *et al.*, "Comparative performability assessment of SFCs: The case of containerized ip multimedia subsystem," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 258–272, 2021.

[7] L. Gu, D. Zengy, J. Hu, B. Liz, and H. Jin, "Layer aware microservice placement and request scheduling at the edge," in *Proc. IEEE INFOCOM*. IEEE, 2021, pp. 1–9.

[8] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. K. Paul, K. Chen, and A. R. Butt, "Large-scale analysis of docker images and performance implications for container storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 918–930, 2021.

[9] A. Kohgadai. (2020) 6 container adoption trends of 2020. [Online]. Available: https://www.stackrox.com/post/2020/03/6-container-adoption-trends-of-2020/

[10] P. Smet, B. Dhoedt, and P. Simoens, "Docker layer placement for on-demand provisioning of services on edge clouds," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 1161–1174, 2018.

[11] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2020–2033, 2019.

[12] Kubernetes. [Online]. Available: https://kubernetes.io/

[13] Y.-Y. Shih, H.-P. Lin, A.-C. Pang, C.-C. Chuang, and C.-T. Chou, "An NFV-based service framework for IoT applications in edge computing environments," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1419–1434, 2019.

[14] R. Kawashima, "Software physical/virtual Rx queue mapping toward high-performance containerized networking," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 687–700, 2021.

[15] T. Goethals, F. DeTurck, and B. Volckaert, "Extending kubernetes clusters to low-resource edge devices using virtual kubelets," *IEEE Transactions on Cloud Computing*, 2020.

[16] L. Cui, Z. Chen, S. Yang, Z. Ming, Q. Li, Y. Zhou, S. Chen, and Q. Lu, "A blockchain-based containerized edge computing platform for the internet of vehicles," *IEEE Internet of Things Journal*, vol. 8, no. 4, pp. 2395–2408, 2021.

[17] S. Guo, K. Zhang, B. Gong, W. He, and X. Qiu, "A delay-sensitive resource allocation algorithm for container cluster in edge computing environment," *Computer Communications*, vol. 170, pp. 144–150, 2021.

[18] J. Zhang, X. Zhou, T. Ge, X. Wang, and T. Hwang, "Joint task scheduling and containerizing for efficient edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 2086–2100, 2021.

[19] Z. Wu *et al.*, "Blender: A container placement strategy by leveraging zipf-like distribution within containerized data centers," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.

[20] M. Niu *et al.*, "Gmta: A geo-aware multi-agent task allocation approach for scientific workflows in container-based cloud," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1568–1581, 2020.

[21] R. Kang *et al.*, "Virtual network function allocation in service function chains using backups with availability schedule," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4294–4310, 2021.

[22] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and evaluation of algorithms for mapping and scheduling of virtual network functions," in *Proc. IEEE NetSoft*. IEEE, 2015, pp. 1–9.

[23] Y. Zhang *et al.*, "Service chain provisioning with sub-chain-enabled coordinated protection to satisfy availability requirements," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.

[24] C. Ren *et al.*, "On efficient service function chaining in hybrid software defined networks," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.

[25] M. S. Castanho, C. K. Dominicini, M. Martinello, and M. A. M. Vieira, "Chaining-box: A transparent service function chaining architecture leveraging bpf," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 497–509, 2022.

[26] J. Zu, G. Hu, D. Peng, S. Xie, and W. Gao, "Fair scheduling and rate control for service function chain in nfv enabled data center," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2975–2986, 2021.

[27] Y. Wang, C.-K. Huang, S.-H. Shen, and G.-M. Chiu, "Adaptive placement and routing for service function chains with service deadlines," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3021–3036, 2021.

[28] N. Siasi, M. Jasim, J. Crichigno, and N. Ghani, "Container-based service function chain mapping," in *2019 SoutheastCon*. IEEE, 2019, pp. 1–6.

[29] Y. Bi *et al.*, "Multi-objective deep reinforcement learning assisted service function chains placement," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4134–4150, 2021.

[30] R. A. Addad, D. L. C. Dutra, T. Taleb, and H. Flinck, "Ai-based network-aware service function chain migration in 5g and beyond networks," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 472–484, 2022.

[31] D. T. Nguyen *et al.*, "Placement and chaining for run-time IoT service deployment in edge-cloud," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 459–472, 2020.

[32] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *Proc. IEEE INFOCOM*. IEEE, 2021, pp. 1–9.

[33] S. Natarajan, A. Ghanwani, D. Krishnaswamy, R. Krishnan, P. Willis, and A. Chaudhary, "An analysis of container-based platforms for NFV," *IETF draft, Apr*, 2016.

[34] Docker, "About storage drivers," accessed Mar. 26, 2022. [Online]. Available: https://docs.docker.com/storage/storagedriver/

[35] X. Wang, C. Wu, F. Le, A. Liu, Z. Li, and F. Lau, "Online VNF scaling in datacenters," in *Proc. IEEE CLOUD*, 2016, pp. 140–147.

[36] L. Wang, M. Dolati, and M. Ghaderi, "Change: Delay-aware service function chain orchestration at the edge," in *Proc. IEEE ICFEC*. IEEE, 2021, pp. 19–28.

[37] A. Chakrabarti, C. Chekuri, A. Gupta, and A. Kumar, "Approximation algorithms for the unsplittable flow problem," *Algorithmica*, vol. 47, no. 1, pp. 53–78, 2007.

[38] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[39] W. Hoeffding, *Probability Inequalities for sums of Bounded Random Variables*, 1994, pp. 409–426.

[40] Gurobi Optimization LLC., "Gurobi Solver," accessed Jan. 2, 2022. [Online]. Available: https://www.gurobi.com

[41] (2022) containerized-sfc. https://github.com/mahdidolati/containerized-sfc. Accessed Mar. 27, 2022.

[42] G. K. Zipf, *Human behavior and the principle of least effort: An introduction to human ecology*. Boston, MA, USA: Addison-Wesley, 1949.