Traffic Prediction-Based VNF Auto-Scaling and Deployment Mechanism for Flexible and Elastic Service Provision

Bo Yi, *Member, IEEE*, Jiacheng Wang, Qiang He, Xingwei Wang, Min Huang, Sajal k. Das, *Fellow, IEEE*, and Keqin Li, *Fellow, IEEE*

***Abstract*—Network Function Virtualization (NFV) provides a flexible way to provision new services by decoupling network functions from hardware and implementing them as Virtual Network Functions (VNFs). However, the rapid development of technologies greatly promotes the explosion of diverse services, which directly results in the exponential increase of heterogeneous traffic. In addition, such a tremendous amount of heterogeneous traffic will generate bursts in a more dynamic and unexpected manner, so it becomes extremely hard to satisfy the customer demands. Aiming at addressing these challenges, this work proposes a positive and elastic VNF deployment mechanism for service provisioning, which introduces three novelties: *1) a Gated Recurrent Unit (GRU) based traffic prediction model is established to predict the unexpected and dynamically changing traffic behaviors in advance with the accuracy over 98%; 2) a closed-loop system is formed, in which the prediction model can learn and evolve continuously to respond to more complex scenarios; 3) different states of VNF are introduced and dynamically switched to deal with the current demands with reduced cost by avoiding frequent VNF initialization and destroy.* The experimental results indicate that the proposed mechanism outperforms the state-of-the-art methods, which include achieving over 98% prediction accuracy, improving the service acceptance rate by more than 18%, and reducing the overall cost by more than 20%.**

***Index Terms*—Elastic VNF deployment, gate recurrent unit, network burst, traffic prediction, VNF cache.**

Manuscript received 12 August 2023; revised 26 February 2024 and 3 July 2024; accepted 29 July 2024. Date of publication 8 August 2024; date of current version 9 October 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500800, in part by the National Natural Science Foundation of China under Grant 62032013, Grant 92267206, and Grant U22A2004, in part by The Fundamental Research Funds for the Central Universities of China under Grant N2316006, and in part by the Liaoning Provincial Science and Technology Joint Program under Grant 2023-MSBA-079. *(Corresponding authors: Xingwei Wang; Qiang He.)*

Bo Yi, Jiacheng Wang, and Xingwei Wang are with the College of Computer Science and Engineering, Northeastern University, Shenyang 110169, China (e-mail: yibobooscar@gmail.com; wangxw@mail.neu.edu.cn).

Qiang He is with the College of Medicine and Biological Information Engineering, Northeastern University, Shenyang 110169, China (e-mail: heqiang cai@gmail.com).

Min Huang is with the College of Information Science and Engineering, Northeastern University, Shenyang 110819, China (e-mail: mhuang@mail.neu.edu.cn).

Sajal k. Das is with the Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65409 USA (e-mail: sdas@mst.edu).

Keqin Li is with the Department of Computer Science, State University of New York, New York, NY 12561 USA (e-mail: lik@newpaltz.edu).

Digital Object Identifier 10.1109/TSC.2024.3440050

I. Introduction

THE rapid development of new technologies such as metauniverse, autopilot, quantum computation, and 6G greatly promotes the appearance of many immersive applications which demand ultra-high bandwidth and low delay to guarantee the service quality. Besides, with over billions of terminals accessing the network, the required service is becoming more and more diverse and complex [1]. In this way, the traditional middlebox-based service provisioning is no longer flexible and efficient enough to handle these new situations. Network Function Virtualization (NFV) [2] appears as a novel network paradigm that decouples the network functionalities from hardware and implements them as Virtual Network Functions (VNFs). Then, NFV provides a more flexible service provisioning way by combining and chaining VNFs with different required functions on-demand [3], which are then referred to as the Service Function Chain (SFC). In particular, an illustration example is given in Fig. 1(a), where the initial SFC is composed of two VNFs of firewall and gateway in sequence.

In fact, there is already a lot of research proposed to study the VNF deployment problem. However, it is aware that most of these work mainly focused on determining where to deploy VNFs and they usually ignored a practical fact that the service provisioning flexibility is achieved by frequently creating newly required and deleting expired VNF instances on demand [4]. The more the number of services, the higher the VNF changing frequency, which then leads to extremely high maintenance costs. Besides, the manual initialization process of VNF instance also takes a long time, which further increases the cost [5].

Taking the above analysis into consideration, we refine the VNF deployment problem and propose a more critical one which we refer to as the VNF auto-scaling problem. Specifically, VNF auto-scaling [6] intends to better optimize the service provisioning process by dynamically scaling out or scaling in the number of VNF instances. For example, the number of VNF instances can be either scaled in to avoid resource waste when the network traffic becomes less or scaled out to quickly respond to sudden bursts in case the network load becomes extremely heavy. However, the problem is that excessive VNF scaling out would increase the maintenance cost and excessive VNF scaling in would interrupt many services and lead to Service Level Agreement (SLA) conflicts. Therefore, 1) how many VNFs

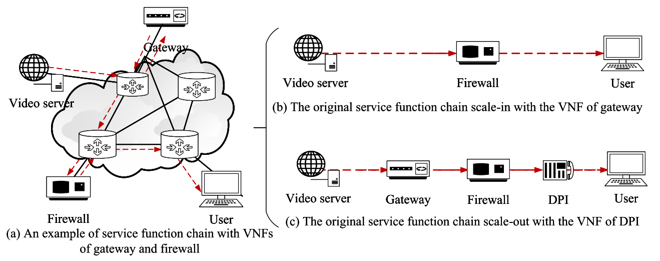

Gateway
Video server
Firewall
User
(a) An example of service function chain with VNFs of gateway and firewall
Video server
Firewall
User
(b) The original service function chain scale-in with the VNF of gateway
Video server
Gateway
Firewall
DPI
User
(c) The original service function chain scale-out with the VNF of DPI

Fig. 1. Illustration of service function chain scenarios.

should be scaled out/in; 2) when to scale out/in; 3) where to scale out/in, become vital important to implement the VNF auto-scaling and deployment mechanism for flexible service provision.

For instance, due to the dynamic requirements of the customer, two cases may happen to the service chain, as shown in Fig. 1. The first one is that customers may no longer need the gateway VNF, such that the gateway VNF is removed from this service chain, which has been illustrated in Fig. 1(b), where only firewall is remained. This is the case of service function chain scale-in. The second one is that customers may need a new VNF of Deep Packet Inspector (DPI) to perform the deep packet inspection function, so that it is added behind the firewall function as shown in Fig. 1(c). This is the case of service function chain scale-out.

Existing solutions addressing the VNF auto-scaling and deployment problem can be separated into two categories, namely, the threshold based reactive VNF scaling [7], [8], [9], [10], [11], [12], [13], [14], [15] and the prediction-based proactive VNF scaling [16], [17], [18], [19], [20], [21], [22], [23], [24], [25]. The former method initiates the VNF scaling operation upon detecting certain performance indicators, which can result in a delayed response to traffic surges. Moreover, determining the most suitable threshold is challenging, often necessitating dynamic adjustments. Consequently, the reactive approach may introduce a lag between the actual event and the scaling action. In contrast, prediction-based VNF scaling strategies are more adept at addressing traffic bursts proactively. They can preemptively allocate resources to scale out VNFs based on predictive analytics. However, it is important to note that these predictive models are typically executed offline and are decoupled from the real-time VNF scaling environment, potentially diminishing the accuracy and effectiveness of their forecasts. Additionally, the convergence time for these models can be quite lengthy.

Targetting on the above challenges, we propose a traffic prediction-based VNF auto-scaling and deployment mechanism with the following contributions:

- We design a Gate Recurrent Unit (GRU) based traffic prediction model which can be used to predict the unexpected and dynamically changing traffic behaviors in advance with high accuracy. Then, the prediction results are used to guide the VNF auto-scaling and deployment process.
- We build a closed-loop VNF auto-scaling and deployment system framework that integrates the GRU based prediction model. In this closed-loop system, the prediction model is able to learn and evolve continuously to respond to more complex scenarios.
- We introduce different VNF states to further optimize the VNF re-deployment cost. By switching the state of the popular and unused VNFs to ready instead of termination for a longer time, a great deal of VNF creation and destruction operation could be avoided and the cost is naturally saved.

The rest of this work is summarized as follows: Section II summarizes the related work. Section III introduces the system framework and problem model. Section IV proposes the traffic prediction-based VNF auto-scaling and deployment mechanism. Section V analyzes the experimental results and Section VI concludes this work.

II. Related Work

Currently, the VNF scaling and deployment problem has been widely studied. Among these state-of-the-art researches, we can divide them into two main categories according to the strategy they adopt, which are the traditional VNF deployment and intelligent VNF deployment.

A. *Traditional VNF Scaling and Deployment*

Extensive research has addressed the VNF deployment problem. Zhao et al. [7] introduced a delay-aware method for a balanced trade-off between reliability and low delay. Qi et al. [8] transformed VNF deployment into a combinational problem. Recognizing the scale of services, Luo et al. [9] transitioned to parallel deployment, addressing challenges with a Viterbi-based algorithm. This concept was further explored in [10], implementing the model by breaking down large flows. Wang et al. [11] proposed a comprehensive solution addressing VNF deployment and service chaining, acknowledging the trade-off's impact on performance. Moving beyond traditional approaches, [12] advocates for a marketplace-based distribution, execution, and lifecycle management of VNFs, enhancing flexibility and efficiency. Similarly, [13] establishes a marketplace dedicated to VNFs, fostering rapid deployment and scalability, promoting flexibility to meet dynamic network service demands, while [14] addresses delay-aware virtual network function placement and routing in edge clouds, emphasizing strategies for minimizing delays.

The VNF deployment is also prevalent in emerging scenarios such as 5G/6G and edge computing. To illustrate, Yang et al. [15] approached VNF deployment in the edge network as a Markov decision process, employing the Behrman iteration strategy to maximize deployment utility. Carlinet et al. [16] revisited this challenge within a 5G/6G environment, simplifying it to the knapsack problem and leveraging traditional solutions. However, with increasing scenario complexity, the demand for intelligence rises. Intelligent VNF deployment is emerging as a trend in response to this. For instance, Lorido et al. [17] employed deep reinforcement learning to tackle the VNF deployment problem, while Sun et al. [18] utilized graph neural learning methods for the same purpose. Nevertheless, it is crucial to acknowledge that these machine learning-based approaches may face challenges in adapting to diverse environments, such as varying topologies, and might exhibit long convergence times.

Traditional VNF deployment is relatively static, but recent efforts focus on elastic VNF deployment, addressing the auto-scaling challenge. Tang et al. [19] used dynamic programming for service chain scaling, while Sun et al. [20] established relationships between new and original VNFs, optimizing deployment. Ma et al. [21] proposed a two-stage process, determining VNF locations in the first stage and chaining them into a service in the second. Additionally, [11] integrates Service Function Chains (SFC) composition, deployment, and allocation to enhance resource usage. [22] introduces an SFC deployment scheme for load balancing and reallocation, improving network service acceptance rates and reducing latency. However, achieving true elastic VNF deployment remains a distant goal.

B. Intelligent VNF Scaling and Deployment

In prior research within the domain of intelligent VNF (Virtual Network Function) deployment, one category has focused on optimizing VNF placement using common machine learning algorithms. For instance, Xu et al. [23] introduced a solution to address VNF deployment using LSTM and DNN. N. Seo et al. [24] discussed methods for updating VNF deployment using reinforcement learning algorithms, emphasizing the use of scaling actions for updates. Besides, S. Park et al. [25] proposed an optimal VNF deployment method based on machine learning, leveraging machine learning techniques for automation and optimization in the VNF deployment process. Some have tried other unconventional networks as well, Kim el al. [26] further explored the optimization of virtual network function deployment based on graph neural networks, emphasizing the importance of intelligent decision-making using graph-structured data, while [27] delves into the application of machine learning techniques for predicting and optimizing Virtual Network Function (VNF) deployment decisions, particularly in the context of dynamically changing network conditions.

Another category of research focuses on optimizing VNF deployment in virtual networks using graph convolution and deep reinforcement learning. Such as Qiu et al. [28] proposed a deployment algorithm for virtual network functions based on graph convolution deep reinforcement learning to optimize VNF deployment in networks and Qu et al. [29] discussed a reliable service function chain deployment method based on deep reinforcement learning aimed at enhancing the reliability of network functions. There are also some of them concentrate on practice usage, [30] emphasized the dynamic deployment of service function chains in SDN/NFV-enabled cloud management systems based on forecasts to achieve efficient resource utilization. Similarly, He et al. [31] explores the integration of deep reinforcement learning and attention mechanisms to optimize virtual network function placement and routing while, Kun et al. [32] explored methods for deploying service function chains with parallelized VNFs under conditions of resource demand uncertainty in mobile edge computing environments. However, in order to fully realize intelligent VNF deployment, much additional research is still needed.

C. Discussion

Although extensive research has been conducted on VNF deployment and auto-scaling by using the traditional or the proactive prediction-based methods, we should be aware that these work either require long-time offline model training or suffer from low VNF utilization. Additionally, they have not taken into consideration the resource fragments that may occur frequently during the dynamic VNF deployment process, thus causing huge resource waste and extremely low resource utilization. All these issues directly leads to high cost and complex operation for VNF constituted service provision. Therefore, we refine the VNF scaling and deployment problem, aiming at which we propose the following VNF auto-scaling and deployment solution.

III. System Architecture and Problem Model

A. System Architecture

In this work, we design an elastic VNF auto-scaling and deployment architecture for flexible service provisioning, which is designed based on three-layer architecture shown in Fig. 2. In particular, we can see that there are three layers in this architecture from bottom up, that is, the infrastructure layer (i.e., NFV infrastructure, NFVI [22]), the management layer and the control layer. Our major contributions are carried out in the control layer so as most of state-of-art work including our benchmark algorithm, where the decisions are made and executed via the REST API to the management layer to finally manage the underlying resources.

Specifically, for the bottom infrastructure layer, it is constructed by virtualizing the discrete physical resource and forming virtual resource pools for isolated and secure resource allocation for flexible service provision. For the middle management layer, it connects the other two layers via the north-bound and south-bound interfaces respectively. After receiving the decision from the control layer, it will first transfer the decision to instructions that can be recognized by the infrastructure layer and then forward these instructions to the bottom layer. To fulfill such function, it is composed of several components including the Resource-Server module for periodic resource state synchronization, the Registry-Service module for VNF image management, the Metrics module for performance monitor, the Prometheus and Alarm modules for data visualization and the Engine module for VNF auto-scaling. As for the top control layer, it first needs to obtain the status of VNF and traffic in the cluster from the engine module, which are then used to train the prediction model. According to the prediction results, we can then deploy the corresponding VNFs reasonably the subsequent VNF elastic deployment module.

Generally, the service requests will arrive continuously. In this condition, the designed system will periodically collect the network information by the Resource-Server module. Then, the raw data, which includes the port, packet received, etc, is sent to the Engine module, where the traffic features are extracted and clustered to train the prediction model. After that, the prediction results are used to guide the VNF auto-scaling process. In this

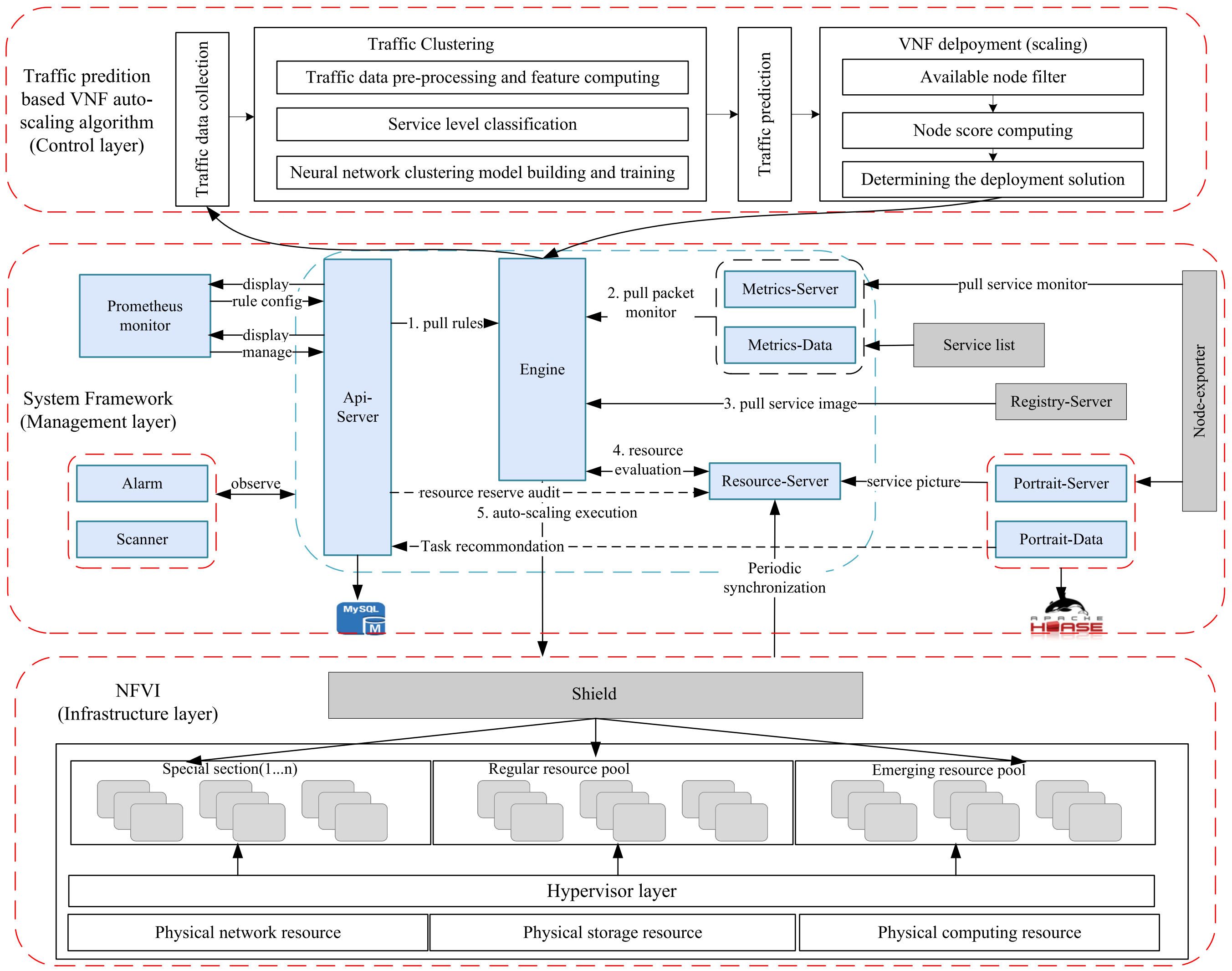


Fig. 2. System architecture.

way, a closed-loop is formed, where the prediction model can learn and evolve continuously.

It is noted that the engine module is the core part of this work since it involves the proposed VNF auto-scaling mechanism. In particular, we can observe that the collected traffic data will go through four stages until the deployment policy is determined and sent back to the Engine to execute.

B. Problem Model

The underlying physical network is defined as an undirected graph $G(V, E, M)$, where V is the set of nodes, E is the set of underlying physical links, and M is the set of VNF instances. Given any node $u \in V$, its memory and CPU core can be denoted by C_u^{mem}, C_u^{core}. The processing delay of this node is d_u. The bandwidth resource of the link (u, v) is denoted by C_{uv}^{bw} and the corresponding link delay is denoted by d_{uv}. Assuming there are P types of VNFs, so that given any VNF $m \in M$, it belongs to one specific type, that is, $type_p^m = \{0,1\}\}$, where $type_p^m = 1$ means m is the p-type VNF, otherwise not. In addition, the CPU resource of this VNF is denoted by C_m^{cpu} and the processing delay of this VNF is d_m which is different in unique VNFs.

Then, a directed graph $G_f = (V_f, E_f)$ is used to represent the VNF forwarding graph, where V_f is the set of VNFs and E_f is the set of logical links between VNFs. Given any service f, its required bandwidth, memory, CPU, and core resource are denoted by $\varphi_f^{bw}, \varphi_f^{mem}, \varphi_{f,p}^{cpu}, \varphi_p^{core}$ respectively. The maximum tolerable delay of f is φ_f^{delay}. Hence, for any service f that exists in the network, all the physical links traversed by it are connected, so that

$$\sum_{v \in V} \sum_{v_f u_f \in E_f} \left(z_{uv}^{u_f v_f} - z_{vu}^{u_f v_f} \right) = \begin{cases} 1, & u \text{ is the entrance} \\ -1, & v \text{ is the exit} \\ 0, & \text{other} \end{cases} \tag{1}$$

where $z_{uv}^{u_f v_f}, z_{vu}^{u_f v_f} \in \{0,1\}$ mean whether f using the logical link (u_f, v_f) traverses the physical link (u, v) or (v, u).

In addition, once a physical link (u, v) is traversed by a service, the two end-nodes of this link are also traversed, and it follows that

$$z_u^{u_f v_f} z_v^{u_f v_f} = \begin{cases} 1, & \text{if } z_{uv}^{u_f v_f} = 1 \\ 0, & \text{otherwise} \end{cases} \quad \forall u_f, v_f \in E_f \tag{2}$$

where $z_u^{u_f v_f}, z_v^{u_f v_f} \in \{0,1\}$ indicate whether f is using the logical link (u_f, v_f) that traverses the physical node u or v.

In order to formulate the object, another two binary variables are defined as follows:

$$z_m^{u_f}, y_u^m \in \{0,1\}, \forall m \in M, u \in V, u_f \in V_f. \tag{3}$$

where $z_m^{u_f}, y_u^m = 1$ mean u_f provides resource for the VNF m and m is deployed on u, otherwise not.

The objective of this work is to minimize the overall cost which is composed of three parts, that is, the deployment cost (denoted by $\mathbb{D}$), the maintenance cost (denoted by $\mathbb{F}$) and the penalty cost by rejecting service requests (denoted by $\mathbb{U}$). In particular, the three kinds of cost can be calculated as follows:

$$\mathbb{D} = c^{place} \sum_{u\in V} \sum_{m \in M} \max\{y_u^m - \hat{y}_u^m, 0\}$$

$$\mathbb{F} = \Delta t \sum_{u\in V} \sum_{m\in M} y_u^m$$

$$\mathbb{U} = \sum_{f \in \vartheta \cup R} c^{penalty} \left(1 - \Phi\left(\sum_{u\in V} z_u^{u_f v_f} > 0\right)\right) \tag{4}$$

where c^{place} is cost of deploying one single VNF. $c^{penalty}$ is the penalty factor of rejecting service requests. Δt is any given time period and $\Phi(*) = \{0,1\}$ indicates whether the service has been successfully provided. In particular, to improve the readability of this work, we summarize all the notations in Table I.

Then, denoting the services arriving, ending and living in the network during the next time period Δt as ϑ, L and R respectively, the object of minimizing the overall cost during Δt is formulated as follows:

$$\begin{aligned}
&\min_{z_{uv}^{u_f v_f}, z_u^{u_f v_f}, z_m^{u_f}} \eta_1 \mathbb{D} + \eta_2 \mathbb{F} + \eta_3 \mathbb{U} \\
\text{s.t.} \; & \sum_{u_f v_f \in E_f} \sum_{f \in \vartheta\cup L \cup R} \varphi_f^{bw} (z_{uv}^{u_f v_f} - \hat{z}_{uv}^{u_f v_f}) \le C_{uv}^{bw} \\
& \sum_{u_f v_f \in E_f} \sum_{f \in \vartheta\cup L \cup R} \varphi_f^{mem} (z_u^{u_f v_f} - \hat{z}_u^{u_f v_f}) \le C_u^{mem} \\
& \sum_{u_f \in E_f} \sum_{f \in \vartheta\cup L \cup R} \varphi_{f,p}^{cpu} (z_m^{u_f} - \hat{z}_m^{u_f}) \le C_m^{cpu} \\
& \sum_{m\in M} n_p^{core} y_u^m \le C_u^{core}, \forall p \in P, u \in V \\
& \sum_{uv\in E} \sum_{u_f v_f \in E_f} d_{uv} z_{uv}^{u_f v_f} + \sum_{u\in V} \sum_{u_f v_f \in E_f} d_u z_u^{u_f v_f} \\
& \quad + \sum_{m\in M} \sum_{u_f \in V_f} d_m z_m^{u_f} \le \varphi_f^{delay} \\
& \sum_{m\in M} \sum_{u_f\in V_f} z_m^{u_f} y_u^m \le z_{uv}^{u_f v_f}, \forall u \in V, \\
& u_f, v_f \in E_f, f \in \vartheta \cup R \\
& \quad \sum_{p\in P} q_p^m = 1, \forall m \in M \\
& (1), (2), (3)
\end{aligned} \tag{5}$$

TABLE I
NOTATIONS

$G(V,E,M)$	the undirected graph of the underlying physical network	$G_f = (V_f, E_f)$	a directed graph of the VNF forwarding graph
V	the set of nodes	V_f	the set of VNFs
E	the set of underlying physical links	E_f	the set of logical links between VNFs
M	the set of VNF instances	f	a service
C_u^{mem}	memory of node u	φ_f^{bw}	the bandwidth required by service f
C_u^{core}	number of CPU core of node u	φ_f^{mem}	the memory required by service f
d_u	the processing delay of node u	$\varphi_{f,p}^{cpu}$	the CPU required by service f
C_{uv}^{bw}	the bandwidth resource of the link (u,v)	φ_p^{core}	the CPU cores required by service f
d_{uv}	the corresponding link delay of the link (u,v)	φ_f^{delay}	the delay of service f
$type_p^m = \{0,1\}$	one specific type(m) VNF in P types	$z_{uv}^{u_f v_f}$	whether service f using the logical link (u_f, v_f) traverses the physical link (u,v)
C_m^{cpu}	the CPU resource of VNF m	$z_{uv}^{u_f v_f}$	whether service f using the logical link (u_f, v_f) traverses the physical link (v,u)
d_m	the processing delay of VNF m	$z_u^{u_f v_f}, z_v^{u_f v_f}$	whether service f is using the logical link (u_f, v_f) that traverses the physical node u and v
$z_m^{u_f}, y_u^m = 1$	if $z_m^{u_f}, y_u^m = 1$ mean u_f provides resource for the VNF m and m is deployed on u, otherwise not.	n_p^{core}	the total number of CPU cores required for the VNF instance.

where $\eta_1 + \eta_2 + \eta_3 = 1$ are weighting coefficients. n_p^{core} represents the total number of CPU cores required for the VNF instance. $\hat{z}_{uv}^{u_f v_f}$, $\hat{z}_u^{u_f v_f}$ and $\hat{z}_m^{u_f}$ are the binary variables in the previous time period before $z_{uv}^{u_f v_f}, z_u^{u_f v_f}, z_m^{u_f}$.

Specifically, the first three constraints stipulate that the bandwidth, memory, and CPU resources allocated during the time interval Δt must not surpass the total available resources. It is important to note that the memory and CPU constraints are node-specific, whereas the bandwidth constraint pertains to the links. The fourth constraint dictates that the number of CPU cores allocated to the VNFs must not exceed the cores available on the respective nodes. The fifth constraint ensures that the actual end-to-end delay for each service does not surpass its maximum tolerable threshold. The sixth constraint mandates the existence of a planned path for each service chain, ensuring that traffic traverses all related VNFs in the correct sequence. Lastly, the seventh constraint specifies that each VNF must belong exclusively to one category.

Nevertheless, the VNF deployment problem has already been proved to be NP-hard [35], which means that it cannot be solved in polynomial time. In this work, the VNF deployment has been developed to jointly consider the VNF auto-scaling, which introduces more constraints and variables under the main considerations of: 1) How many VNF are required; 2) Where to deploy these VNFs; 3) When to deploy the VNFs. With such complex considerations and constraints, solving this ILP model in acceptable time becomes not even possible, such that we propose a more efficient mechanism to jointly address the VNF auto-scaling and deployment problem.

IV. Traffic Prediction-Based VNF Auto-Scaling and Deployment Mechanism

A. Traffic Data Collection and Pre-Processing

The core of the proposed mechanism is the prediction model which relies heavily on the dataset used to train this model. Hence, to guarantee the prediction performance of this model, we need to continuously collect the most updated traffic data from network and use them to train the prediction model. The collected traffic data are denoted by a three-tuple $(port, tx_{bytes}, t)$, where $port$ is the port monitored, tx_{bytes} is the bytes sent out and t is the time. Hence, given any two times $t_1, t_2 (t_1 < t_2)$, the traffic rate during this period can be calculated as $f_{rate} = \frac{tx^2_{bytes} - tx^1_{bytes}}{t_2 - t_1}$.

Generally, the typical way to collect the network traffic relies on using the network monitors which inspect the network in different granularities (e.g., flow-level and packet-level). However, a common way is to minitor the network status periodically, which may easily miss the important traffic feature data, due to the fixed periods. In this regard, we dynamically change the monitoring period according to the current traffic rate in network, in case of missing some important data samples. Then, denoting the average traffic rate by f_{avg} and the max/min monitoring cycle by $T_{\max}, T_{\min}$, the current monitoring cycle should be

$$T_{monitor} = \min \left\{ \frac{|f_{avg}|}{|f_{rate}|} T_{\min}, T_{\max} \right\} \tag{6}$$

After network monitor, we can get the original data which are then used for model training. However, these raw data need to be pre-processed before using. In this way, we first transform the collected traffic data into the matrix format. Namely, we first denote that the traffic rate f_{rate} from node i to j at time t by f^t_{ij}. Then, the traffic matrix is expressed as $TM^t = (f^t_{ij})_{n*n}$ where $n \in [1, |V|]$ is the number of nodes used.

However, the traffic between different sources and destinations varies greatly. In order to avoid the deviation caused by such difference and improve the generalization ability of the prediction model, the following normalization process is executed on the traffic data matrix.

$$f^{t'}_{ij} = \frac{f^t_{ij} - \min\{f^t_{ij} | i, j \in [1, n]\}}{\max\{f^t_{ij} | i, j \in [1, n]\} - \min\{f^t_{ij} | i, j \in [1, n]\}}. \tag{7}$$

where $f^{t'}_{ij} \in [0, 1]$.

Executing the above equation for every element in TM^t, we obtain the normalized traffic matrix denoted by $TM^{t'} = (f^{t'}_{ij})_{n*n}$.

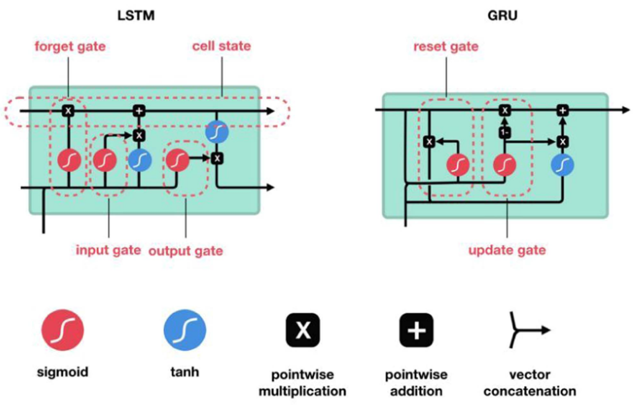


Fig. 3. Structure comparison between LSTM and GRU.

B. Traffic Prediction Model Establishment

The traffic prediction model is built upon the Gated Recurrent Unit (GRU) model, a specialized form of deep learning Recurrent Neural Networks (RNNs). The GRU model is designed to enhance the capture of dependencies across various time scales. When compared with the Long Short-Term Memory (LSTM) model, another variant of RNN, the GRU model simplifies the gating mechanism and neuron state, leading to a significant reduction in the training time required for the network. LSTM networks are known for their inclusion of an input gate, a forget gate, and an output gate. These gates regulate the flow of information, enabling the network to selectively retain or discard past information. In contrast, the GRU model, recognized as one of the most potent deep learning models for time series prediction, streamlines the LSTM's structure by replacing the input and forget gates with a single update gate. This modification results in a more straightforward architecture. For ease of comparison, the structures of both models are illustrated and contrasted in Fig. 3.

Since the input data for the prediction model are already obtained (i.e., $TM^{t'}$), we next define the time series model to train and adjust the prediction parameters. According to the structure of GRU, the relationship between its input and output is expressed as follows:

$$h^l = \sigma \left(W^l h^{l-1} + b^l \right), \tag{8}$$

where h^l, W^l, b^l represent the hidden state output, weight, and bias of the l-th layer in GRU, σ is the activation function.

Then, we denote the training parameter set of GRU by $\Gamma = (W_r, W_z, W_h, b_r, b_z, b_h)$, where the subscripts r, z, h represent the reset gate, update gate and output gate in GRU respectively.

Now, the training objective is to minimize the Mean Square Error (MSE) of this model to obtain the optimal parameter set $\varGamma$. To fulfill this target, the MSE function is defined based on the loss function in the back propagation, as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\left(\tilde{f}_{ij}^{t} - f_{ij}^{t}\right)^{2}. \tag{9}$$

By calculating all the loss, we obtain the training cost as follows:

$$Cost = \frac{1}{k}\sum_{q=1}^{k} MSE_{q}, \tag{10}$$

where q is the number of iterations.

However, the back propagation in GRU needs to calculate the mapping between the model training cost to the gradient of each layer, so that we formulate the relationship between the training cost and the training parameter vector as follows:

$$Cost = \sum_{i=1}^{N}\left(y_i - (wx_i + b)\right)^{2}, \tag{11}$$

where x_i, y_i are the input and output of this model in each layer.

Next, we define the gradient vector as follows:

$$\nabla Cost = \left(\frac{\partial Cost}{\partial w_1}, \frac{\partial Cost}{\partial w_2}, \ldots, \frac{\partial Cost}{\partial w_N}\right)^{T} \tag{12}$$

where $W = w_1, w_2, \ldots, w_N$ is the weight of each layer of the model training cost $Cost$, and the corresponding gradient calculation is as follows:

$$\delta^{l} = \frac{\partial Cost}{\partial a^{l}} \odot \sigma'\left(z^{l}\right) \tag{13}$$

$$\delta^{l} = \left(\left(w^{l+1}\right)^{T}\delta^{l+1}\right) \odot \sigma'\left(z^{l}\right) \tag{14}$$

$$\frac{\partial Cost}{\partial w_{jk}^{l}} = a_{k}^{l-1}\delta_{j}^{l} \tag{15}$$

$$\frac{\partial Cost}{\partial b_{j}^{l}} = \delta_{j}^{i} \tag{16}$$

where δ^{l} is the output error. (14) shows the detailed back propagation process of the output error calculation. (15) and (16) represent the gradient of the cost function $Cost$.

By continuously applying the chain derivation rule to the above calculation, the model learns all optimal parameters, which are expressed as follows:

$$\hat{\varGamma} = \arg_{\varGamma}\min \frac{1}{n^{2}}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\left(\tilde{f}_{ij}^{t} - f_{ij}^{t}\right)^{2} \tag{17}$$

Moreover, we use the Adam optimizer to optimize the training cost, since it can efficiently and automatically adjust the training step size. This optimization process is shown as follows:

$$\nabla Cost_t = \nabla_{\varGamma} f_t(\varGamma_{t-1}) \tag{18}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot \nabla Cost_t \tag{19}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2)\cdot \nabla Cost_t^{2} \tag{20}$$

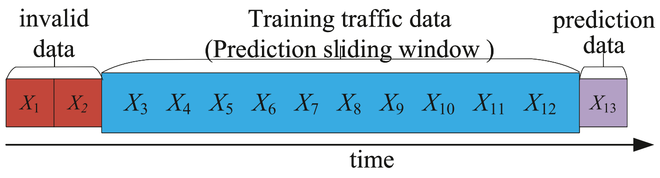

invalid data
Training traffic data (Prediction sliding window)
prediction data
X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8 X_9 X_{10} X_{11} X_{12} X_{13}
time

Fig. 4. Prediction sliding window.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^{t}} \tag{21}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^{t}} \tag{22}$$

$$\hat{\varGamma}_t = \hat{\varGamma}_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon}\hat{m}_t \tag{23}$$

where $\nabla Cost_t$ represents the gradient vector of the cost function $Cost$. $f_t(\varGamma)$ represents the stochastic optimization objective function of the parameters. m_t, v_t represent the mean and square values of the gradient. $\beta_1, \beta_2 in[0,1)$ represent the exponential decay rates. η, ε are the learning rate and a constant.

Based on the above process, we can obtain the prediction model. However, if the error between the prediction result and the actual result is large, we go back to (8) to continue the training until a high prediction accuracy is achieved.

C. Traffic Prediction

According to the established prediction model, for any network with n nodes, the traffic in this network at time t can be expressed by the traffic matrix $TM^{t} = (f_{ij}^{t})$, where f_{ij}^{t} indicates the traffic from the node i to the node j at the time t. Hence, given T time periods before t, we can get the history traffic matrixes, that is, $TM^{T} = (TM^{t}, TM^{t-1}, TM^{t-2}, \ldots, TM^{t-T})$, where TM^{T} is the set of history traffic.

Usually, the prediction is carried out based on historical data. In this work, the traffic prediction is also implemented based on the history traffic, i.e., TM^{T}. Therefore, we introduce the prediction sliding window in this work. For example, as shown in Fig. 4, the size of the prediction sliding window is 10, so that the traffic in the latest 10 cycles (i.e., from $X3$ to $X12$) are used to predict the traffic in the next cycle (e.g., $X13$). It is easy to note that the history data of $X1$ and $X2$ are not used to train the prediction model, because it has been used before and not in the currently sliding window. Training the prediction model with $X1$ and $X2$ will not achieve much progress, but the training time and computing resource consumption are inevitable.

Apparently, the size of the sliding window may affect the overall performance greatly. In this way, we set the prediction sliding window to T accordingly and the value of T becomes vital important. However, too large window size can easily lead to extremely long training time, while too small window size would also lead to low prediction accuracy performance. Therefore, we determine the size of this prediction sliding window according to the following evaluating results. Next, we introduce the vector function to transform the traffic matrix into a vector as the input of the prediction model, so that given any TM^t, we have

$$Vector(TM^t) = \left(f_{11}^t, f_{12}^t, \ldots, f_{ij}^t, \ldots, f_{n(n-1)}^t, f_{nn}^t\right), \tag{24}$$

where the position of f_{ij}^t in this vetor is calculated according to the expression $(i * n + j)$.

In this way, TM^T is changed into the expression of $(Vector(TM^t), Vector(TM^{t-1}), \ldots, Vector(TM^{t-T}))$ which is then regarded as the input of the GRU based prediction model to predict the traffic (i.e., TM^{t+1}) of the next cycle. Besides, we should be aware that the obtained prediction results are also saved as the history training data.

Then, based on the above definition, the traffic prediction can be simplified as: given the input $(Vector(TM^t), Vector(TM^{t-1}), \ldots, Vector(TM^{t-T}))$ and the GRU based traffic prediction model, we can output $Vector(TM^{t+1})$ for the next cycle.

D. VNF Deployment and Auto-Scaling

Based on the prediction model, we can estimate the traffic condition in the near future, which can be transformed into conclusions that *1) what types of VNFs are required in the network; 2) how many the VNFs of the types are required to carry the load in the next time period; 3) where to deploy these VNFs.* Specifically, given any VNF to be deployed, we first need to determine the potential candidate host nodes which must satisfy the following basic constraints:

- The state of this node must be "ready" so that it can be used to provide the VNF service. In addition, we should be aware that there are also other states such as pending and termination in case of different situations.
- The CPU/memory resource of this node should exceed the required amount of resource to deploy the VNF, so as to guarantee the service quality provided.
- The end-to-end delay should not exceed the maximum tolerable value when using this node to deploy the required VNF, since processing delay will be generated on this node, which contributes to the final end-to-end delay.

Since the VNF deployment usually needs multiple kinds of resources (e.g., CPU and memory), we next score the candidate nodes to create the priority. In this work, the score of each node is calculated based on the linear combination of different resources' utilization. The object is to minimize the cost and maximize the accepted services, so that the smaller the resource utilization of this node, the higher the probability this node would be selected. Similarly, the score is calculated based on resource utilization, so the smaller the value of the score, the higher the node priority.

However, the linear combination is a common method used to cluster different parameters, which leads to the fact that the aggregation policy should be carefully designed. In this work, we try to calculate the score for each node by using the weighted aggregation of resource states. Then, it follows that

As explained, these decisions are given to the Engine to execute automatically, so that the VNF auto-scaling is implemented.

$$score = \sum_{i=1}^{|P|} p_i w_i, \tag{25}$$

where $score$ is the weighted score. P is the set of resource parameters and p_i is the state parameters of the i-th kind of resource. w_i is the weight of the i-th kind of resource, which determines the importance of this kind of resource.

Actually, to accept more services, the distribution of network load should be fair and the VNF deployment should also be fair among different nodes to avoid large fluctuations of resource utilization. In this way, the weights attached to the different resources are not static, but dynamically change after each VNF deployment. In this case, the weight parameters can reflect the distribution of resources when the node is running. The greater the fluctuation of the weight, the higher the probability that the corresponding resource will contribute more to the final score. However, this situation will destroy the balance among these resources and reduce the opportunity for this node to accept more requests. Therefore, given the parameter p_i, it includes θ states, then the corresponding weight is calculated as follows:

$$\begin{aligned} w_i &= \frac{\sigma_i^2}{\mu_{i,\theta}} \\ \text{where } \mu_{i,\theta} &= \frac{1}{\theta}\sum_{k=1}^{\theta} x_k \\ \sigma_{i,\theta}^2 &= \frac{1}{\theta}\sum_{k=1}^{\theta}(x_k - \mu_{i,\theta})^2, \end{aligned} \tag{26}$$

where x_k indicates the k-th state of p_i. $\mu_{i,\theta}, \sigma_{i,\theta}^2$ are the mean and variance values of the θ states.

Since the size of the historical data set grows at a constant rate, the amount of calculation is also increasing with a much more fast speed (i.e., $O(p \cdot n^2)$). The typical challenge is that when some new traffic data arrive, traditional methods need to carry out the repeated calculation for all the historical data, which is time-consuming and meaningless. In order to solve this challenge, all the calculation is performed online in this work and we introduce the mean recursive estimation method to calculate the mean and variance for new traffic data incrementally, as follows:

Since the size of the historical traffic data grows at a constant rate, the amount of calculation to get the parameter in equation (27) is also increasing with a much more fast speed (i.e., $O(p \cdot n^2)$). The typical challenge is that when some new traffic data arrive, traditional methods need to carry out the repeated calculation for all the historical data, which is time-consuming and meaningless. In order to solve this challenge, all the calculation is performed online in this work and we introduce the mean recursive estimation method to calculate the mean and variance for new traffic data incrementally, as follows:

$$\mu_{i,\theta} = \mu_{i,\theta-1} + \frac{(x_\theta - \mu_{i,\theta-1})}{\theta}$$

$$\sigma_{i,\theta}^2 = \frac{\sigma_{i,\theta-1}^2(\theta-1) + (x_\theta - \mu_{i,\theta-1})(x_\theta - \mu_{i,\theta})}{\theta}. \tag{27}$$

Combining (25) and (26), we can update the parameter weights dynamically. In particular, it is aware that the parameter weights are updated in the runtime and we only need to do calculations for the new traffic data, so that the computing power is saved and the efficiency is improved. In addition, one more advantage is that we can keep the amount of calculation almost the same during the system running time so that the influence that the parameter weights updating process has on the VNF deployment is stable. Based on the above process, the node having the highest score will be selected as the node to deploy the corresponding VNF.

Due to the situation that the network environment changes dynamically, we need to take the VNF auto-scaling into consideration so as to adapt to such changing environments and requirements. Typically, the VNF auto-scaling includes the vertical scaling (i.e., increase or decrease the VNF resource) and the horizontal-scaling (increase or decrease the number of VNF instances). However, the vertical scaling will inevitably lead to the reboot of VNF instances and cause SLA violation, so that we mainly focus on implementing the VNF auto-scaling in the horizontal direction.

In addition, as explained, the threshold based VNF scaling may fail to serve the short-term services, so the prediction-based proactive VNF auto-scaling is carried out. Now, given the predicted result TM^{t+1}, we first need to extract two critical factors which are 1) what types of VNFs are required; 2) how many of these VNFs are required. By analyzing the statistics of the prediction traffic, we can easily obtain the traffic processing capacity required for any type of VNF at time t+1. Let's denote one type of the required VNFs by f_j with the rate denoted by $f_{j,rate}$, and the total required number is H, so that we have

$$\overline{P}_j(t+1) = \sum_{i=1}^{H} f_{j,rate}^i, \tag{28}$$

where $\overline{P}_j(t+1)$ indicates the processing capacity required by the j-th type of VNF at the time t+1.

Based on the above equation, the number of the new added j-th type of VNF is expressed as follows:

$$q_j^{new}(t+1) = \begin{cases} \left\lceil \frac{\overline{P}_j(t+1) - \overline{P}_j(t)}{C_j} \right\rceil, & \text{if } \overline{P}_j(t+1) > \overline{P}_j(t) \\ 0, & \text{otherwise.} \end{cases} \tag{29}$$

where $q_j^{new}(t+1)$ indicates how many new j-th type of VNF should be deployed and C_j indicates the average processing capability of f_j. $\overline{P}_j(t)$ indicates the processing capacity required by the j-th type of VNF at the time t.

There are two cases in (28) should be noted, that is: 1) when $\overline{P}_j(t+1) > \overline{P}_j(t)$, we should deploy $q_j^{new}(t+1)$ new VNFs to handle the extra traffic load. 2) when $\overline{P}_j(t+1) \le \overline{P}_j(t)$, the number of the j-th type of VNFs in network is enough to process the traffic arriving at the time t+1.

Despite this, when $\overline{P}_j(t+1) << \overline{P}_j(t)$, it means that the existing VNFs are excessive redundant. Directly deleting these VNFs will lead to high deployment costs, since some deleted VNFs may be demanded after that. In this case, we propose a dynamic VNF state switching strategy to maximize the utilization of VNFs. For example, some temporarily unused but popular VNFs can be set to ready state for a relative long time instead of terminating them directly. Due to its popular characteristics, it may be used shortly, such that frequent creation and destroy of the VNFs could be avoided, which then reduce the cost and improve the resource utilization. Since $\overline{P}_j(t+1)$ is far smaller than $\overline{P}_j(t)$, we define the number of the redundant VNFs as follows:

$$q_j^{del}(t+1) = \left|\overline{P}_j(t+1) - \overline{P}_j(t)\right|. \tag{30}$$

Instead of deleting all the $q_j^{del}(t+1)$ VNFs immediately, we first set their states to "waiting" with a waiting time threshold (denoted by $threhold_j^{time}$), so as to cache them for a period of time. In this state, these VNFs will consume very little resource. Then, if the redundant VNFs are needed again, their states will be set to "running" from "waiting" and the $threhold_j^{time}$ is ignored. Otherwise, the corresponding VNF will be deleted. Apart from reducing the cost, this dynamic VNF state switching strategy can also reduce the service delay, because the new VNF deployment also consumes time.

V. Performance Evaluation

A. Setup

The experiments are carried out using the Python language on a high-performance server PowerEdge R740 with the Ubuntu 18.04 with kernel 5.15.0. We create five Virtual Machines (VM) to simulate the server cluster environment, where one VM is used as the master node to execute the algorithm and the rest four VMs are used as working nodes to provide physical resources. In particular, Kubernetes v1.25.2 [36], [37] with docker v20.10.23 is deployed as the service provisioning kernel on the master node and Python 3.8 is used to host the traffic prediction model. The experimental section of the paper is based on the CERNET backbone network.

Taking the practical scenarios into consideration, we set the number of VNF types to 15, which can almost cover all the common network function requirements in real-world scenarios, which include the Firewall, Deep Packet Inspector (DPI), Intrusion Detection System (IDS), Intrusion Prevention System (IPS), Network Address Translation (NAT), Domain Name System (DNS), Broadband Network Gateway (BNG), Web Proxy (WP), 4K/8K video server, Content Delivery Network (CDN), IP Multimedia Subsystem (IMS), coding and decoding server, Load Balancer (LB), Dynamic Host Configuration Protocol (DHCP) server and OpenVSwitch (OVS). We use these VNF as individual service instead of SFC. Since each VNF is implemented in a container, we set the required memory and CPU to [1, 200]M and [1, 200]MHz respectively, following the uniform distribution, where the number of VNF that may increase or decrease with the rate that is set between [0.5, 3] following the uniform distribution. As for each of the VNF constituted service funtion chains, it demands the CPU resource between [1, 40]MHz following the uniform distribution and the

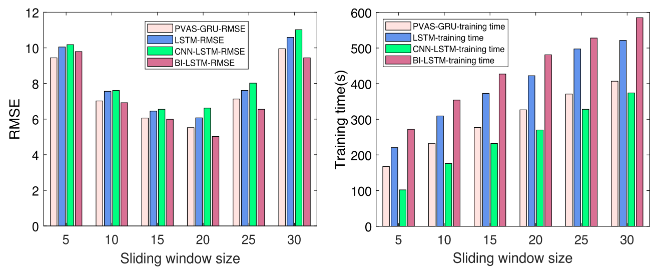
TABLE II
PARAMETERS

Parameter	Distribution	Value
number of VNF types	none	15
memory required to instantiate a VNF	uniform distribution	[1,200]M
CPU required to instantiate a VNF	uniform distribution	[1,200]MHz
VNF increase/decrease rate	uniform distribution	[0.5,3]
CPU required per service request	uniform distribution	[1,40]MHz
bandwidth required per service request	uniform distribution	[1,30]Mbps
service request generation time	poisson distribution	50 requests per 100 time units (mean)
service life cycle	exponential distribution	1000 time units (mean)
maximum tolerable delay	none	1000 time units
node CPU	none	32GHz
node memory	none	64GB

bandwidth resource between [1, 30]Mbps following the uniform distribution as well. The service requests are generated following the Possion distribution with a mean speed of 50 requests per 100 time units. Besides, the life cycle of each service follows an exponential distribution with a mean value of 1000 time units. The maximum tolerable delay of each service is set to 1000 time units. Finally, the CPU resource of each node is set to 32GHz and the memory resource is set to 64G. These important parameters are summarized and shown in Table II.

B. Benchmarks and Metrics

In order to verify the performance of our work, we select 1) LSTM, CNN-LSTM, Bi-LSTM to compare with the designed GRU prediction model; 2) two state-of-the-art methods to compare with the proposed VNF auto-scaling and deployment mechanism, which are the one proposed by Takaya et al. [34] (Threshold-based VNF elastic deployment mechanism, TVD) and the one proposed by Tang et al. [19] (Dynamic network function instance scaling mechanism based on traffic forecasting and VNF Placement, DTVP). In particular, the first one is a reactive VNF scaling method which will execute the scaling operation only when detecting the situation which the load threshold is exceeded. On the other hand, the second one uses a sliding window linear regression model taking the historical traffic data as input. This regression model fits the historical traffic data to generate a linear equation, which is then used to predict future traffic trends. Additionally, to ensure high service availability, DTVP accounts for prediction errors by incorporating a standard deviation adjustment factor based on recent traffic fluctuations and calculating an upper bound for traffic forecasts. This approach not only adapts to dynamic traffic changes, but also optimizes resource utilization while maintaining service quality. For simplicity, the proposed method is referred to as the Prediction-based VNF Auto-Scaling (PVAS).



(a) RMSE (b) training time

Fig. 5. RMSE and training time.

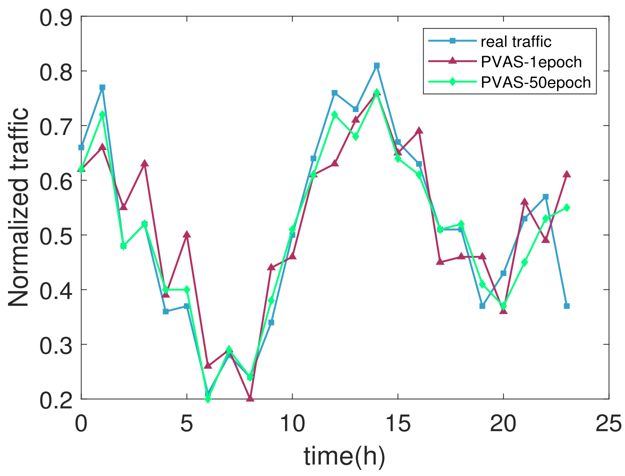


Fig. 6. Traffic prediction.

The performance metrics include 1) the Root Mean Square Error (RMSE) and the training time for prediction performance comparison; 2) the average network utilization, CPU utilization, delay, VNF load strength, service acceptance rate, and number of active VNFs for network performance comparison.

C. Results

1) Prediction Results: We first evaluate the performance of the proposed traffic prediction model and the corresponding results are shown in Figs. 5 and 6. In particular, Fig. 5 provides the results of RMSE and the training time with 50 epochs against different sizes of sliding windows. Apparently, it is easy to note that 1) the RMSE increases first and then decreases for the two prediction models,it is because initially, due to the increase in window size, the model is able to see longer historical information, resulting in a decrease in RMSE. However, as the window size continues to increase, it may introduce too much redundant information during training, which can interfere with the model, and so the sliding window size for achieving the best RMSE is the turning point, that is, 20; 2) the training time per epoch grows with the increase of the sliding window size; 3) PVAS-GRU outperforms LSTM in terms of both the RMSE and the training time. 4) Although CNN-LSTM has a faster training time compared to GRU, both of them could be accepted in our actual demand, besides, its RMSE (Root Mean Square Error) is higher than GRU, which is more important in our experiment.

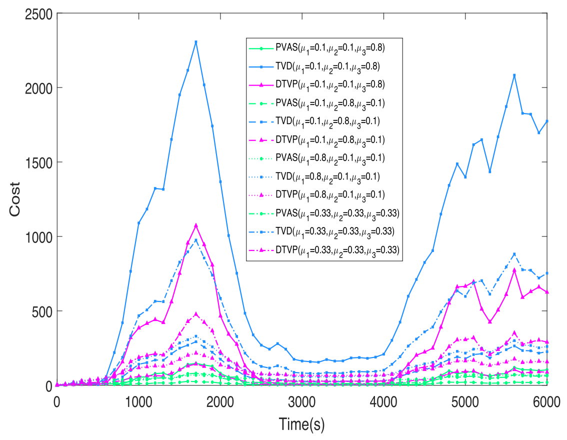
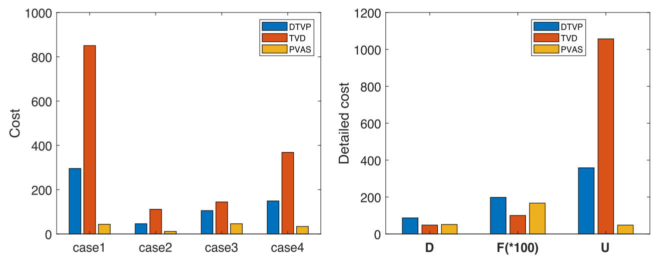


Fig. 7. Overall cost.



(a) Average cost in four cases (b) Detailed cost of $\mathbb{D}, \mathbb{F}, \mathbb{U}$

Fig. 8. Average cost and detailed sub-cost.

As for Bi-LSTM, it has a low RMSE, but its training time is too long to meet the requirement of early deployment in this experiment. Therefore, it is also not applicable.

Generally, the time spent on training the prediction model increases with the increase of the sliding window size, because the larger the sliding window, the more data will be used to train the model and the longer the training time. In addition, observing the training time in detail, we can notice that it increases linearly before the sliding window size exceeds 20 and then the increasing trend becomes slow after that. It has two main reasons. 1) As training progresses, the model may start to converge, meaning that the updates to the model's parameters become smaller. This can result in a slowdown in the rate of increase of training time, as fewer computational resources are needed for fine-tuning. 2) Larger windows may contain more data redundancy, which could reduce the model's need to learn new information. This can lead to a decreased rate of growth in training time, as the model becomes more efficient at handling the increased data volume without significant additional computational effort. Now, combined with the conclusion that the best RMSE is achieved when the sliding window size is 20, we set the size of the sliding window to 20 for the rest experiments in this work.

For the results in Fig. 6, they are achieved based on the normalized network traffic data. In particular, there are three lines that indicate the actual network traffic and the predicted traffic with 1 and 50 epoch-training respectively. From this figure, we can see that the predicted traffic with 50 epoch training is relatively closer to the actual network traffic. More specifically, we calculate the similarity between the real traffic and the predicted one according to the cosine similarity theory $\frac{\mathbf{ab}}{||\mathbf{a}||||\mathbf{b}||}$, which actually reflects the prediction accuracy, that is, 99.6% with 50 epochs and 98.7% with 1 epoch. In this case, we can see that there are still improvement from 1 epoch to 50 epochs. By comparing the quantified data, the predicted traffic achieved by the model with 50-epoch training is much more similar to the real traffic, which means that the 50-epoch training based prediction model has a higher prediction accuracy.

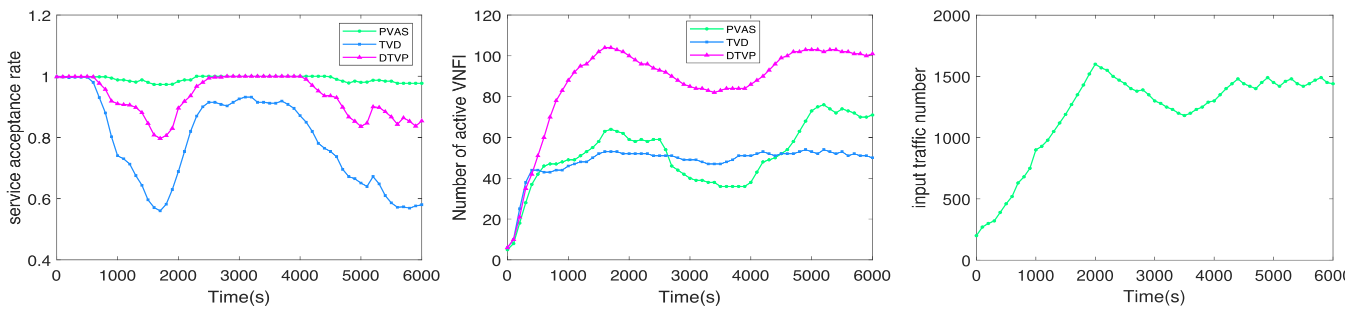
Moreover, let's review the results in Fig. 6, we can notice that the predicted traffic of PVAS-50 epoch is exactly the same as the practical traffic at the times of 2, 3, 6, 7, 8, 10, 17, 18, 20 and 22. Meanwhile, the predicted traffic of PVAS-50 epochs at other times also approaches the real traffic. In this case, we observe that the prediction accuracy will not increase greatly when the number epoch exceeds 50, but the training time still increases. Therefore, the training epochs are set to 50.

2) Performance Results: Now, based on the setting of the sliding window size and the training epoch, we use the webbench tool [38] to inject real service traffic into the network, which follows the Possion distribution. As explained, the overall cost consists of the VNF deployment cost, maintenance cost and the penalty cost and the corresponding weights for the three sub-costs are μ_1, μ_2, μ_3 respectively. Then, we change their weights to provide four testing cases, that is, case1, $\mu_1 = 0.1, \mu_2 = 0.1, \mu_3 = 0.8$; case2, $\mu_1 = 0.1, \mu_2 = 0.8, \mu_3 = 0.1$; case3, $\mu_1 = 0.8, \mu_2 = 0.1, \mu_3 = 0.1$; case4, $\mu_1 \approx 0.33, \mu_2 \approx 0.33, \mu_3 \approx 0.33$.

The overall cost achieved under the four cases are shown in Fig. 7, where we can see that 1) the cost does not increase or decrease linearly, but follows the Possion distribution, which is in accordance with the injected real traffic, so that this can reflect the correctness of our experiments in a certain extent; 2) given the same weight setting, the proposed method achieves lower cost in any case; 3) the average cost achieved by different methods is the lowest at case2, which means that the weight for the maintenance cost is higher and this is exactly in accordance with the practical situation.

In order to clearly observe the cost, we further show the details in Fig. 8, where Fig. 8(a) summarizes the average cost in four cases and Fig. 8(b) shows the detailed cost of $\mathbb{D}, \mathbb{F}, \mathbb{U}$. For the average results, we can see that 1) the proposed method has the lowest cost in the four cases; 2) the proposed method has very lower $\mathbb{D}, \mathbb{U}$ and relatively higher $\mathbb{F}$, which is due to the case that the dynamic VNF state switching mechanism in this work allows us to significantly reduce the deployment cost and penalty cost with only a small increase in maintenance cost; 3) the reduction of $\mathbb{D}, \mathbb{U}$ achieved by PVAS is far more larger than the increase of $\mathbb{F}$ achieved by PVAS. Therefore, taking the above into consideration, PVAS achieves the best performance in terms of reducing the overall cost.

Next, we evaluate the service acceptance rate as well as the number of active VNFs used to serve the traffic. The corresponding results are summarized and shown in Fig. 9, where Fig. 9(a) calculates the real-time service acceptance rate; Fig. 9(b) shows



(a) Service acceptance rate (b) Number of active VNFs (c) input traffic trend

Fig. 9. Service acceptance rate and active VNFs.

the number of active VNFs that are used to carry these service requests at the same time; and Fig. 9(c) shows the corresponding traffic trend. From the overall perspective, we can see that the service acceptance rate is on the opposite against the traffic amount, while the number of active VNFs is proportional to the amount of traffic. This is reasonable, because on one hand, more traffic means the network is in a congestion situation, which naturally prevents from accepting more services. On the other hand, more traffic means that more VNF would be created to support such demands.

Specifically, for the results in Fig. 9(a), it is easy to note that the proposed method has the highest service acceptance rate. Meanwhile, the proposed method keeps a stable and high service acceptance rate which is almost 98%. Comparatively, the other two benchmarks achieve about 80% (DTVP) and 60% (TVD) service acceptance rates on average. Such a difference is due to the response strategy provided by the three methods. On one hand, the proposed PVAS has a very high prediction accuracy and can provide the future traffic trend along the time, so that we can deploy the required VNFs in the correct place in advance in a very fine-grained way. However, for TVD, it can only react when the traffic load exceeds the threshold. As for DTVP, it predicts the average future traffic load which is actually a value and may not be able to reflect the details at any time in the future. In contrast, the neural network approach used in this paper can achieve a higher resolution by adjusting the parameter T. In addition, compared with the other two methods, we build the system that can provide the proposed method with continuous and updated data to learn and evolve in the long run. This is also a very critical factor for the proposed method to achieve a higher service acceptance rate.

In addition, let's see the results of the active VNFs in Fig. 9(b), where we can observe a similar trend as shown in Fig. 9(a). Actually, the number of active VNFs can reflect the deployment cost to a certain extent, because the more active VNFs, the higher the deployment cost would be. During the time of [0,500], the network traffic load is small (resource is enough), so the three methods need almost the same VNFs to provide services. However, with the increase of traffic load between the time of [500, 1700], the number of active VNFs deployed by the three methods becomes different. First, for DTVP, we can see that it creates a lot of active VNFs compared with TVD and PVAS. Despite this, DTVP does not take the VNF load balance into consideration, so the service rejection would also happen especially when the network load becomes high. On the other hand, the proposed method is quite suitable for processing service requests in such a high load environment, so that it can accept the arriving service requests using a small number of active VNFs. During the time of [1700,3500], the network traffic load begins to decrease and we can see that the number of active VNFs deployed by the two prediction-based method (i.e., PVAS and DTVP) also decreases, while TVD still remains stable. When the time exceeds 4000, the number of active VNFs deployed by PVAS and DTVP increases obviously, because they need more VNFs to carry the upcoming traffic. Although PVAS and DTVP can both adapt to the network load, it is easy to note that PVAS can fulfill this target with less number of active VNFs.

Next, we evaluate the average traffic rate (i.e., throughput), CPU utilization, average delay and average VNF load against different numbers of service requests in Fig. 10, where Fig. 10(a) shows the average traffic rate, Fig. 10(b) shows the CPU utilization, Fig. 10(c) shows the average delay and Fig. 10(d) shows the VNF load. Let's first look at the average traffic rate in 10(a), we can see that PVAS achieves a higher traffic rates than TVD and DTVP. Apparently, different numbers of service requests have different impacts on the average traffic rate. In particular, when the number of parallel requests reaches 3500, the average traffic rate of PVAS approaches 500 Mbps, while TVD achieves 445 Mbps and DTVP achieves 460 Mbps. Although PVAS and DTVP are both prediction-based proactive auto-scaling methods, DTVP predicts the average traffic rate in the future, which may not be applied to the situation of large-scale service requests (e.g., over 4000 requests). Besides, PVAS also introduces the dynamic VNF state switching strategy to optimize the VNF utilization and accept more requests. Thus, PVAS improves the average traffic rate by about 5.64%~8.41% than DTVP and about 9.84%~18.49% than TVD.

For the CPU utilization in Fig. 10(b), the higher the better, since it means less time to complete a task. By calculation, the average CPU achieved by the three methods are 43.967% (PVAS), 37.998% (TVD) and 40.554% (DTVP). Obviously, PVAS has the highest CPU utilization and it achieves about 14.23%~16%, 7.57%~9.95% higher CPU utilization than TVD and DTVP respectively.

In Fig. 10(c) and (d), it is clearly that average delay is increasing first until the number of requests reach a threshold individual

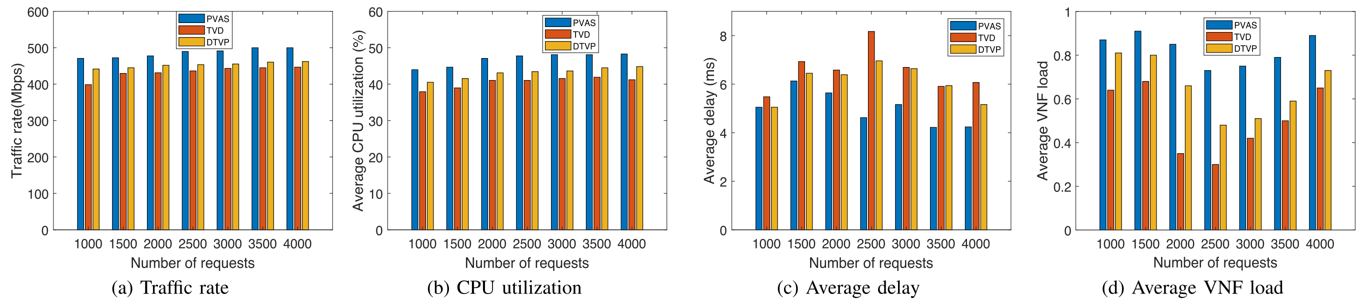


Fig. 10. Results of resource utilization, delay and VNF load.

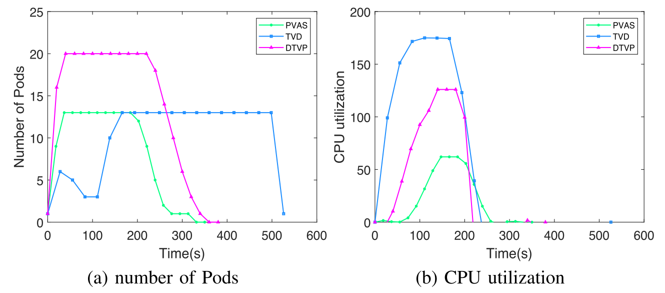


Fig. 11. Detailed number of Pods and CPU utilization.

for three chosen algorithms, and decreasing until reaching a more or less stable point, which is in the contrast for average VNF load. This is because when the number of request streams increases, the three algorithms will automatically scale up the service VNFs to meet the demand of the request streams. As a result, initially, when the number of request streams is not large, the latency increases as the number of request streams grows. However, when the number of request streams exceeds a certain threshold, due to the scaling of the VNFs, both the latency and the load on the VNFs actually decrease as the number of request streams continues to increase.

For the average delay in Fig. 10(c), three types of algorithm used in the experiments PVAS still outperforms the other methods. Another phenomenon is that the delay advantage achieved by PVAS becomes bigger when the number of requests increases. For example, when the number of requests is 1000, the average delay of PVAS, DTVP and TVD are 5.07 ms, 5.06 ms and 5.5 ms respectively, so the delay gap may be [0,0.43] ms. However, when the number of service requests is 4000, the average delay of PVAS, DTVP and TVD are about 4.15 ms, 5.2 ms and 6 ms respectively, so that the delay gap becomes [1.05, 1.85] ms. For the average VNF load in Fig. 10(d), PVAS achieves the largest VNF load. Reviewing the results in Fig. 10(b), we are aware that PVAS has a smaller number of active VNFs. Jointly taking the two conditions into consideration, we can come to the conclusion that PVAS can maximize the VNF utilization to carry more traffic with fewer VNFs.

Lastly, we inject a short-term burst of traffic into the network to observe how PVAS, TVD and DTVP respond. The corresponding results are summarized in Fig. 11, where Fig. 11(a) and (b) include the relation between number of Pods, CPU utilization and time respectively In particular, several phenomena can be observed: 1) the CPU utilization achieved by PVAS does not exceed 100% during the burst time, while that of TVD and DTVP both exceed 100% (i.e., 175% and 126%), which means that PVAS balances the network, while TVD and DTVP make the network overloaded when the traffic burst arrives. Besides, the overload time caused by TVD is much longer. 2) PVAS and DTVP can scale the number of pods in advance to adapt to the network traffic load based on the prediction results. Taking TVD as the example, since it is a threshold based reactive method, the time when the CPU utilization exceeds 100% is exactly the time that network overload occurs, that is, 1 minute 45 seconds after the burst. Let's see the data of PVAS and DTVP, we can see that they both scale the number of pods at the time of 45 seconds and 30 seconds after the burst. 3) PVAS scale the number of pods to 13 for the rest burst time, so that PVAS can process this traffic burst with 13 pods and the maximum CPU utilization is 62%. However, DTVP needs 20 pods to carry this burst and the maximum CPU utilization is 126%. From the above observations, we can conclude that PVAS can adapt to the burst traffic easily, which makes it performing better than the other benchmarks.

VI. Conclusion

A. Summary

The rapid development of new network technology promotes the explosion of diverse services, which directly results in the exponential increase of heterogeneous traffic. Thus, we propose a traffic prediction-based VNF auto-scaling and deployment mechanism for provisioning flexible services and meeting the needs of the dynamically changing environment. In particular, the GRU model is applied in the proposed solution, so as to form a closed-loop system among the traffic collection, GRU training and prediction, policy execution over network with the accuracy over 98%. Experimental results indicate that the proposed mechanism outperforms the other state-of-the-art methods. Despite this, we should be aware that the prediction accuracy is achieved under the virtual environment, where the container is used to provide VNF constituted services and virtual machine is used as the host. In addition, the shortest path algorithm is used to chain the traffic along different VNFs without optimization, which means that the traffic may take longer time to traverse the corresponding VNFs along this path, thus leading to larger probability of failures and costs. Nevertheless, the routing path optimization is not the main focus of this work. Hence, the future work include exploring the potential performance of the proposed solution in real-world scenarios especially with the optimization of the traffic routing path.

B. Threats to Validity

However, in this study, we also recognize the existence of factors that may threat the validity of the results. The first is related to some I/O components within the architecture proposed in this paper, if they are stateful services, they will have a significant impact on end-to-end latency, such as the database that appears in the architecture. In the experiment part of our paper, we use stateful service since our huge amount of data. However, if it could be replaced with some simpler stateless services on the actual experimental platform, it will make the end-to-end latency more stable. In addition to this, most of the VNFs used in the experiment of this paper are homogeneous services at the L2/L3 layer because of some practical factors, which may limit the generalizability of the experimental results when applied to more complex network environments that include different types of VNFs. In future experiments, a variety of VNFs of different types and levels should be used to enhance the generalizability of the results as much as possible.

References

[1] W. Jiang, B. Han, M. A. Habibi, and H. D. Schotten, “The road towards 6G: A comprehensive survey,” *IEEE Open J. Commun. Soc.*, vol. 2, pp. 334–366, 2021.

[2] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, “Network function virtualization: Challenges and opportunities for innovations,” *IEEE Commun. Mag.*, vol. 53, no. 2, pp. 90–97, Feb. 2015.

[3] S. Yang, F. Li, S. Trajanovski, R. Yahyapour, and X. Fu, “Recent advances of resource allocation in network function virtualization,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 295–314, Feb. 2021.

[4] Y. Wu, W. Zheng, Y. Zhang, and J. Li, “Reliability-aware VNF placement using a probability based approach,” *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 3, pp. 2478–2491, Sep. 2021.

[5] H. Yu et al., “Octans: Optimal placement of service function chains in many-core systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2202–2215, Sep. 2021.

[6] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, and B. Mukherjee, “Auto-scaling VNFs using machine learning to improve QoS and reduce cost,” in *Proc. IEEE Int. Conf. Commun.*, 2018, pp. 1–6.

[7] Y. Zhao et al., “Service function chain deployment for 5G delay sensitive network slicing,” in *Proc. Int. Wirel. Commun. Mobile Comput.*, 2021, pp. 68–73.

[8] S. Qi et al., “Energy-efficient VNF deployment for graph-structured SFC based on graph neural network and constrained deep reinforcement learning,” in *Proc. 22nd Asia-Pacific Netw. Operations Manage. Symp.*, 2021, pp. 348–353.

[9] J. Luo, J. Li, L. Jiao, and J. Cai, “On the effective parallelization and near-optimal deployment of service function chains,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1238–1255, May 2021.

[10] M. Wang, B. Cheng, S. Wang, and J. Chen, “Availability- and traffic-aware placement of parallelized SFC in data center networks,” *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 182–194, Mar. 2021.

[11] Z. Wang, J. Zhang, T. Huang, and Y. Liu, “Service function chain composition, placement and assignment in data centers,” *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 4, pp. 1638–1650, Dec. 2019.

[12] L. Bondan et al., “FENDE: Marketplace-based distribution, execution, and life cycle management of VNFs,” *IEEE Commun. Mag.*, vol. 57, no. 1, pp. 13–19, Jan. 2019.

[13] G. Xilouris et al., “T-NOVA: A marketplace for virtualized network functions,” in *Proc. IEEE Eur. Conf. Netw. Commun.*, 2014, pp. 1–5.

[14] S. Yang, F. Li, S. Trajanovski, X. Chen, Y. Wang, and X. Fu, “Delay-aware virtual network function placement and routing in edge clouds,” *IEEE Trans. Mobile Comput.*, vol. 20, no. 2, pp. 445–459, Feb. 2021.

[15] Y. Yang, Q. Chen, and G. Zhao, P. Zhao, and L. Tang, “The stochastic-learning-based deployment scheme for service function chain in access network,” *IEEE Access*, vol. 6, pp. 52406–52420, 2018.

[16] Y. Carlinet, E. Gourdin, and N. Perrot, “The Offline Virtual Network Function Packing Problem,” in *Proc. 24th Conf. Innov. Clouds Internet Netw. Workshops*, 2021, pp. 151–158.

[17] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” *J. Grid Comput.*, vol. 12, pp. 559–592, 2014.

[18] P. Sun, J. Lan, J. Li, Z. Guo, and Y. Hu, “Combining deep reinforcement learning with graph neural networks for optimal VNF placement,” *IEEE Commun. Lett.*, vol. 25, no. 1, pp. 176–180, Jan. 2021.

[19] H. Tang, D. Zhou, and D. Chen, “Dynamic network function instance scaling based on traffic forecasting and VNF placement in operator data centers,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 3, pp. 530–543, Mar. 2019.

[20] C. Sun, J. Bi, Z. Meng, T. Yang, X. Zhang, and H. Hu, “Enabling NFV elasticity control with optimized flow migration,” *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2288–2303, Oct. 2018.

[21] W. Ma, O. Sandoval, J. Beltran, D. Pan, and N. Pissinou, “Traffic aware placement of interdependent NFV middleboxes,” in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.

[22] H. Hawilo, M. Jammal, and A. Shami, “Network function virtualization-aware orchestrator for service function chaining placement in the cloud,” *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 643–655, Mar. 2019.

[23] Z. Zaman, S. Rahman, and M. Naznin, “Novel approaches for VNF requirement prediction using DNN and LSTM,” in *Proc. IEEE Glob. Commun. Conf.*, 2019, pp. 1–6.

[24] N. Seo et al., “Updating VNF deployment with scaling actions using reinforcement algorithms,” in *Proc. 23rd Asia-Pacific Netw. Operations Manage. Symp.*, 2022, pp. 1–4.

[25] S. Park, H. G. Kim, J. Hong, S. Lange, J. -H. Yoo, and J. W. -K. Hong, “Machine learning-based optimal VNF deployment,” in *Proc. 21st Asia-Pacific Netw. Operations Manage. Symp.*, 2020, pp. 67–72.

[26] H. G. Kim et al., “Graph neural network–based virtual network function deployment optimization,” *Int. J. Netw. Manage.*, vol. 31, no. 6, 2021, Art. no. e2164.

[27] S. Lange, H. -G. Kim, S. -Y. Jeong, H. Choi, J. -H. Yoo, and J. W. -K. Hong, “Predicting VNF deployment decisions under dynamically changing network conditions,” in *Proc. IEEE 15th Int. Conf. Netw. Service Manage.*, 2019, pp. 1–9.

[28] R. Qiu et al., “Virtual network function deployment algorithm based on graph convolution deep reinforcement learning,” *J. Supercomput.*, vol. 79, no. 6, pp. 6849–6870, 2023.

[29] H. Qu, K. Wang, and J. Zhao, “Reliable service function chain deployment method based on deep reinforcement learning,” *Sensors*, vol. 21, no. 8, 2021, Art. no. 2733.

[30] J. Zhang, Y. Liu, Z. Li, and Y. Lu, “Forecast-assisted service function chain dynamic deployment for SDN/NFV-enabled cloud management systems,” *IEEE Syst. J.*, vol. 17, no. 3, pp. 4371–4382, Sep. 2023.

[31] N. He et al., “Leveraging deep reinforcement learning with attention mechanism for virtual network function placement and routing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 4, pp. 1186–1201, Apr. 2023.

[32] K. Zhang et al., “Towards deploying SFC with parallelized VNFs under resource demand uncertainty in mobile edge computing,” *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 35, 2023, Art. no. 101619.

[33] Google Kubernetes, 2014. [Online]. Available: https://kubernetes.io/docs/

[34] T. Miyazawa, H. Harai, Y. Yokota, and Y. Naruse, “Sparse regression model to predict a server load for dynamic adjustments of server resources,” in *Proc. 22nd Conf. Innov. Clouds Internet Netw. Workshops*, 2019, pp. 249–256.

[35] H. Xuan, S. Wei, Y. Feng, D. Liu, and Y. Li, “Bi-level programming model and algorithm for VNF deployment with data centers placement,” *IEEE Access*, vol. 7, pp. 185760–185772, 2019.

[36] M. Barletta et al., “Criticality-aware monitoring and orchestration for containerized industry 4.0 environments,” *ACM Trans. Embedded Comput. Syst.*, vol. 23, no. 1, pp. 1–28, 2024.

[37] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, “Introducing k4. 0s: A model for mixed-criticality container orchestration in industry 4.0,” in *Proc. IEEE Int. Conf. Dependable Autonomic Secure Comput.-Int. Conf. Pervasive Intell. Comput.- Int. Conf. Cloud Big Data Comput.-Int. Conf. Cyber Sci. Technol. Congr.*, 2022, pp. 1–6.

[38] WebBench, 1997. [Online]. Available: https://github.com/EZLippi/WebBench

Bo Yi (Member, IEEE) is currently a lecturer of computer science and engineering with Northeastern University of China. His research interests include service computing, routing, virtualization, cloud computing in SDN, NFV, DetNet, etc. He has authored and coauthored more than 20 journal and conference articles on *IEEE Transactions on Cloud Computing*, *IEEE Communications Letter*, *IEEE Access*, *Computer Networks*, etc. He is currently the reviewer of *IEEE Communications Survey & Tutorial*, *Communications Letter*, *Computer Networks*, *Journal of Network and Computer Applications*, etc.

Jiacheng Wang is currently working toward the master's degree under the guidance of Yi Bo from Northeastern University of China. His main research field is service optimization in containers and GPU virtualization.

Qiang He received the PhD degree in computer application technology from Northeastern University, Shenyang, China, in 2020. From 2018 to 2019, he was a visiting PhD researcher with the School of Computer Science and Technology, Nanyang Technical University, Singapore. He has authored or coauthored more than ten journal articles and conference papers. His research interests include social network analytic, machine learning, data mining, and software defined networking.

Xingwei Wang received the BS, MS, and PhD degrees in computer science in 1989, 1992, and 1998, respectively from the Northeastern University, Shenyang, China, where he is currently a professor with the College of Computer Science and Engineering. His research interests include cloud computing and future Internet, etc. He has published more than 100 journal articles, books and book chapters, and refereed conference papers. He has received several best paper awards.

Min Huang received the BS degree in automatic instrument, in 1990, MS degree in systems engineering, in 1993, and the PhD degree in control, in 1999 theory from the Northeastern University, Shenyang, China, where she is currently a professor with the College of Information Science and Engineering. Her research interests include modeling and optimization for logistics and supply chain system, etc. She has published more than 100 journal articles, books, and refereed conference papers.

Sajal k. Das (Fellow, IEEE) is the chair of Computer Science Department and Daniel St. Clair Endowed chair with the Missouri University of Science and Technology, Rolla. During 2008–2011, he served the US National Science Foundation as a program director with the Division of Computer Networks and Systems. His current research interests include wireless and sensor networks. He published more than 600 research articles in high quality journals and refereed conference proceedings. He received ten Best Paper Awards in prestigious conferences such as ACM MobiCom99, IEEE PerCom06 and IEEE SmrtGridComm12. He serves as the founding editor-in-chief of the Pervasive and Mobile Computing journal, and as associate editor of *IEEE Transactions on Mobile Computing*, *ACM Transactions on Sensor Networks*, and several others.

Keqin Li (Fellow, IEEE) is a SUNY distinguished professor of computer science with the State University of New York. He is also a National distinguished professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyberphysical systems, heterogeneous computing systems, Big Data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored more than 840 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently an associate editor of the *ACM Computing Surveys* and the *CCF Transactions on High Performance Computing*. He has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Services Computing*, and *IEEE Transactions on Sustainable Computing*.