Joint Request Updating and Elastic Resource Provisioning With QoS Guarantee in Clouds

Gongming Zhao, *Member, IEEE*, Jingzhou Wang, *Graduate Student Member, IEEE*, Hongli Xu, *Member, IEEE*, Yangming Zhao, *Member, IEEE*, Xuwei Yang, and He Huang, *Member, IEEE, ACM*

***Abstract*— In a commercial cloud, service providers (*e.g.*, video streaming service provider) rent resources from cloud vendors (*e.g.*, Google Cloud Platform) and provide services to cloud users, making a profit from the price gap. Cloud users acquire services by forwarding their requests to corresponding servers. In practice, as a common scenario, traffic dynamics will cause server overload or load-unbalancing. Existing works mainly deal with the problem by two methods: *elastic resource provisioning* and *request updating*. Elastic resource provisioning is a fast and agile solution but may cost too much since service providers need to buy extra resources from cloud vendors. Though request updating is a free solution, it will cause a significant delay, resulting in a bad users' QoS. In this paper, we present a new scheme, called real-time request updating with elastic resource provisioning (TRUST), to help service providers pay less cost with users' QoS guarantee in clouds. In addition, we propose an efficient algorithm for TRUST with a bounded approximation factor based on progressive-rounding. Both small-scale experiment results and large-scale simulation results show the superior performance of our proposed algorithm compared with state-of-the-art benchmarks.**

***Index Terms*— Cloud computing, elasticity, request updating, resource provisioning.**

I. Introduction

DRIVEN by the rapid growth of the demand for flexible and efficient computation power, cloud computing has gained much attention from both industry and academia [2]. Compared with building IT infrastructure, cloud computing liberates us from cumbersome tasks (*e.g.*, managing and maintaining devices). Consequently, cloud computing has gained enormous economic earnings in recent years thanks to its convenience and efficiency, and an increasing number of enterprises/individuals are outsourcing their services/workloads to clouds [3].

In practice, there are mainly three roles in clouds: cloud vendors, service providers and cloud users [4]. Cloud vendors (*e.g.*, Amazon Web Services and Google Cloud Platform) are responsible for building and maintaining physical servers. They lease a certain amount of VMs/containers to each service provider. Then, service providers can implement their services like security [5], storage [6], auditing [7], and so on, also called the *Network Function* (NF) instances. Cloud users can purchase the services according to their requirements, so their requests can be scheduled to the corresponding NFs. For instance, a VPN service provider rents several VMs from a cloud vendor to implement a VPN function. Then, cloud users can buy the VPN service from the service provider and acquire services by forwarding their traffic to the corresponding VPNs.

In a large-scale multi-tenant cloud, it is evident that the amount of traffic is tremendous and traffic dynamics is a long-standing problem [3], causing load-unbalancing and even overload among NFs, which will severely affect service availability and execution efficiency [8]. Consequently, it will decrease users' QoS [9]. To deal with traffic dynamics in clouds and ameliorate the QoS, request updating has been widely adopted, which means transferring the requests to another available NF. Existing works usually design the request updating schemes for NF load-balancing [10], [11] or minimizing the makespan [12], [13]. For example, the authors in [10] present a distributed request scheduling mechanism so as to achieve load balancing among servers in data centers by fairly distributing the traffic from the edge switches. The work [13] tries to find an optimal solution for less cloud resource cost under the deadline constraint via the immune-based particle swarm optimization algorithm [14].

Although plenty of works have designed reasonable request updating schemes to handle traffic dynamics, there exist three fundamental disadvantages in request updating. First, the controller capacity is limited. The processing speed of a controller depends on different factors, like the kind of controllers and controller placement. For instance, according to the testing results in Table II of [15], it takes 71.2ms to send sending 1000 flow-mod messages to reroute flows on an ONOS controller, with a network topology consists of 81 switches. As a comparison, authors in [16] also test the update delay and show that ONOS can process 40K flow-mod messages per second, with a topology consisting of 32 switches. For the same topology, the authors in [16] also demonstrate that an OpenDayLight, which is also a main-stream controller

Manuscript received 31 October 2022; revised 17 April 2023; accepted 13 May 2023; approved by IEEE/ACM Transactions on Networking Editor T. He. Date of publication 5 June 2023; date of current version 16 February 2024. This work was supported in part by the National Science Foundation of China (NSFC) under Grant 62102392, in part by the National Science Foundation of Jiangsu Province under Grant BK20210121, in part by the Hefei Municipal Natural Science Foundation under Grant 2022013, and in part by the Youth Innovation Promotion Association of Chinese Academy of Sciences under Grant 2023481. Some preliminary results of this paper were published in the Proceedings of IEEE INFOCOM 2022 [DOI: 10.1109/INFOCOM48880.2022.9796788]. *(Corresponding author: Hongli Xu.)*

Gongming Zhao, Jingzhou Wang, Hongli Xu, and Yangming Zhao are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, China, and also with the Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, Jiangsu 215123, China (e-mail: gmzhao@ustc.edu.cn; wjzwgh@mail.ustc.edu.cn; xuhongli@ustc.edu.cn; zhaoym.ustc@gmail.com).

Xuwei Yang is with Huawei Cloud Computing Technology Company Ltd., Beijing 100085, China (e-mail: yangxuwei1@huawei.com).

He Huang is with the School of Computer Science and Technology, Soochow University, Suzhou, Jiangsu 215123, China (e-mail: emil.hhuang@gmail.com).

Digital Object Identifier 10.1109/TNET.2023.3276881

used in SDN, can only process 0.3K flow-mod messages per second. Thus, generating a large number of new rules for request updating will significantly increase the response time of the controller, making the controller become the bottleneck of the network [17]. Consequently, the controller may be congested and block new arrival requests, leading to the degradation of user experience. Second, request updating involves the delay on switches. The authors in [18] point out that the duration between sending the update messages and updating the corresponding entries on switches may be tens of milliseconds. The delay on switches will lead to wrong operations of flows because of the outdated entries [18]. Finally, there exists state inconsistency issue when updating requests. The work [19] reveals that the update delays on each switch are heterogeneous, and the asynchronous updating actions will cause partial execution of rule-update, which leads to network state inconsistency. This inconsistency is observable in terms of sending packets in transient loops [20], increasing link load [21], or bypassing important waypoints such as a flow classifier [22]. Furthermore, since NFs need to record the states of processed requests, request updating will bring extra delay and overhead to maintain the requests' state consistency on NFs [23]. Because of these three disadvantages, request updating is hard to assure users' QoS in large-scale clouds, and alternative solutions as supplementary methods are urgently needed.

By enabling the virtualization technology, elastic resource provisioning has become a new trend to deal with traffic dynamics in clouds [24], [25], [26], [27], [28]. Elastic resource provisioning means that the system can add or remove resources (such as CPU cores, memory, VM or container instances) to adapt to load variation in a real-time manner [28]. In practice, the cloud vendors usually would offer several suitable configuration types associated with fixed resource combinations [25] and each NF will choose one of the resource configuration types to serve requests from cloud users. For instance, in Google Cloud Platform (GCP) [29], the price for a virtual machine with 4 CPU cores and 15 GB memory is \$97 per month and the price for a VM with 8 CPU cores and 30 GB memory is \$197 per month. The previous works about elastic resource provisioning mainly focus on how to improve network performance [30], increase resource capacity [31] or save the energy [32]. For instance, the authors in [30] provide automatic deployment and proactive scaling of multiple simultaneous web applications methods to improve the infrastructure performance, which has been deployed in Amazon Elastic Compute Cloud.

The idea of elastic resource provisioning holds an excellent promise of solving the problem of traffic dynamics. Compared with request updating solutions, resizing a VM/container only takes tens of milliseconds [25], [33] and there is no need to worry about the problem of request state consistency. Accordingly, users' QoS can be guaranteed. However, it will increase the cost of service providers since they should pay more to cloud vendors for the extra resources. In contrast, with the request updating method, service providers do not have to pay the extra money, but users' QoS may be affected, due to the update delay and the requirement of request state consistency. Hence, we find the two approaches can be complementary to each other to help service providers pay less cost with users' QoS guarantee.

In this paper, we propose real-time request updating with elastic resource provisioning (TRUST) when facing traffic dynamics. Specifically, since request updating would increase the update delay and decrease users' QoS, we try to finish the updating operation under the time threshold T, which will be determined by users' QoS demand. For instance, in a cellular communication network, the authors in [34] show that when the delivery delay is below 150ms, the quality of propagation can be still guaranteed. Meanwhile, we try to minimize the infrastructure cost of purchasing cloud resources. In a nutshell, TRUST can enlighten a way that helps service providers to spend less money with users' QoS guarantee. The main contributions of this paper can be summarized as follows:

1) We comprehensively analyze the current methods to deal with traffic dynamics in clouds and show the advantages and weaknesses of request updating and elastic resource provisioning.
2) We give the formulation of real-time request updating with elastic resource provisioning (TRUST), which can assure users' QoS and save money for service providers. To our best knowledge, this is the first work that takes advantages of both elastic resource provisioning and request updating to handle traffic dynamics.
3) We present a randomized-rounding based algorithm. The performance analysis shows that our algorithm can achieve the optimal value with a high probability.
4) We conduct small-scale experiments and large-scale simulations using real-world topologies and datasets to show that the proposed algorithm can achieve superior performance compared with the state-of-the-art solutions.

II. Preliminaries

A. Commercial Cloud Model

A typical commercial cloud mainly consists of three parts: cloud vendors, service providers and cloud users. Cloud vendors (1) host a set of physical machines, (2) construct and maintain the VMs/containers upon the physical servers, and (3) sell or lease resources to service providers and users. Cloud vendors set several configuration types of VMs/containers, each associated with a certain amount of computing resources (*e.g.*, CPU or RAM). Of course, they will charge a reasonable price for the resources service providers use. Service providers will rent resources from cloud vendors such as a VM with 4 core CPUs and 8G RAM to implement their services, *e.g.*, VPN and ELB, also called *Network Function (NF)* instances. Service providers sell these services to cloud users and process the requests from them. Service providers also need to pay the infrastructure prices to cloud vendors, making a profit from the price gap. Obviously, the service providers all wish to spend less money on infrastructure costs and provide services to users with high QoS guarantee.

We use $S = \{s_1, s_2, \ldots s_{|S|}\}$ to denote the set of physical servers maintained by cloud vendors, and each server s comes with limited resources $R(s)$. Here $R(s)$ can be expanded into a resource vector to represent different types of resources on server s, such as CPU, RAM and bandwidth. Let $C = \{c_1, c_2, \ldots c_{|C|}\}$ represent the set of configuration types, and each configuration type c comes with resources usage $r(c)$, processing capacity $p(c)$ and infrastructure cost $m(c)$. Note that, $r(c)$ can also be expanded into a resource vector. $N = \{n_1, n_2, \ldots n_{|N|}\}$ represents the set of NFs and N_s denotes

TABLE I

CONFIGURATION TYPES AND PRICES ACCORDING TO GOOGLE CLOUD PLATFORM [29]. CONFIGURATION TYPE A CONTAINS A 4 CORE CPU AND 15G RAM AND IS CAPABLE TO HANDLE 50GBPS REQUESTS. CONFIGURATION TYPE B CONTAINS A 8 CORE CPU AND 30G RAM AND IS CAPABLE TO HANDLE 100GBPS REQUESTS

Type	Configuration	Price	Capacity
A	4 core CPU 15G RAM	\$97	50Gbps
B	8 core CPU 30G RAM	\$197	100Gbps

the set of NFs on the physical server $s \in S$. We use $\Gamma = \{\gamma_1, \gamma_2, \ldots \gamma_{|\Gamma|}\}$ to denote the set of requests generated by cloud users. Each request γ comes with a traffic size of $f(\gamma)$, which can be acquired by collecting flow statistics information on the controller [35].

B. A Motivating Example

This section presents a motivating example to demonstrate the advantages of TRUST compared with existing solutions.

As shown in Table I, we assume that cloud vendors will provide two kinds of NF configuration types. A video streaming service provider owns three NFs, denoted as NF_1, NF_2 and NF_3, all initialized as type A to provide high quality online videos. NFs with type A are able to deal with 50 Gbps traffic load. Now the service provider is facing traffic dynamics. The current traffic loads of three NFs are 65 Gbps, 56 Gbps and 27 Gbps, respectively, which means overload on both NF_1 and NF_2. Suppose that every user is enjoying a similar video streaming service and one request takes 10 Mbps traffic load on average. Then, we assume that transferring 100 requests from one NF to another will cost one unit timespan. To guarantee the users' QoS, the service provider needs to finish the updating process within 10 units. Otherwise, the delay may upset the users. The performance comparison of the three algorithms is shown in Table II.

Existing works mainly deal with traffic dynamics by two methods: Elastic Resource Provisioning (ERP) and Request Updating (RU). ERP (*e.g.*, [28], [36]) will buy extra resources to cope with the burstiness traffic. It is an agile solution and will not cause a decrease in users' QoS, but may be expensive. Specifically, in this example, ERP needs to upgrade the configuration to type B for both NF_1 and NF_2. This way will not involve significant update delay and the problem of request state consistency, but cost the service provider an extra 200 USD per month.

RU (*e.g.*, [37], [38]) mainly transfers the requests from the overloaded NF to another available one to alleviate the traffic dynamics. It is a free solution but may spend a lot of time on request updating. Specifically, RU needs to update $1500/600$ requests from NF_1/NF_2 to NF_3, and will cost 21 units timespan, which means RU cannot well satisfy the QoS demand of users. Moreover, RU may not be able to hold all the traffic under some extreme situations due to the limited capacity of NFs. Under this circumstance, the service provider has to abandon or deny some requests.

In practice, since a small delay may not affect the QoS of users remarkably [39], we try to combine the two methods. As a result, service providers will spend less money and still guarantee the QoS. We introduce TRUST, a brand-new approach that combines these two methods. If TRUST is adopted in this example, we only need to transfer 600 requests from NF_2 to NF_3, and upgrade the configuration type of NF_1. Compared with RU, TRUST can finish the updating process within 10 units timespan and will not cost too much compared with ERP. This example fully demonstrates the advantages of TRUST.

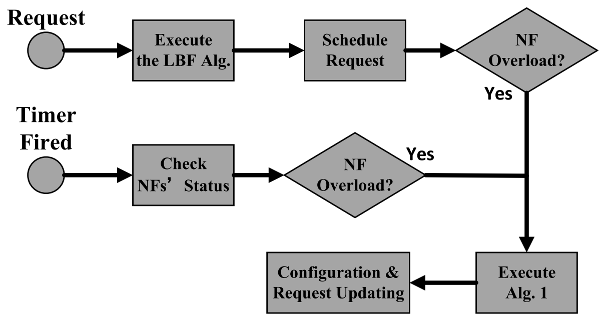


Fig. 1. The system workflow of trust.

C. System Workflow

In this section, we show how our proposed TRUST scheme works in the real production environment. The workflow is shown in Fig. 1. When a new request arrives, it will be scheduled by a greedy based online scheduling algorithm called Least-Burden First (LBF). We choose the NF with the least burden to handle the request. If the chosen NF exceeds the load ratio threshold, Algorithm 1 will be triggered to jointly upgrade the configuration and update the requests. Meanwhile, considering that the traffic is dynamic, we also set a timer (*e.g.*, 10 minutes) to detect NFs' load status periodically. If some NFs are overloaded, Algorithm 1 will be triggered. Till then, we need to estimate the current traffic information based on traffic matrix [40], [41], [42], [43]. We should note that the traffic information is needed only when Algorithm 1 is triggered.

In practice, the two events are highly unlikely to take place at the exact same time. That's to say, they are likely to take place in sequence. For instance, if the first event occurs earlier, we will execute Algorithm 1. Then, the system will enter a *busy waiting* status. In specific, if Algorithm 1 is repeatedly triggered, it needs to check whether there exists Algorithm 1 running. If so, the next running needs to wait until the current execution is over. In conclusion, Algorithm 1 will not be executed twice at the same time under any circumstance.

The reasons why we do not execute updating algorithm for every upcoming request mainly include the following aspects. First, for every request, request updating will bring extra delay and overhead to maintain the requests' state consistency on NFs [23]. For example, after request updating, if a user's requests are scheduled to an NF without maintaining the requests' state, it will incur request state inconsistency and wrong operations. Second, for a controller, the burden to execute the algorithm will severely affect the performance. If we adopt the request updating in an online manner, the controller needs to frequently execute the proposed algorithm to acquire the decisions, which may take a long-time span, causing significant delay. As a result, the controller will be overwhelmed for handling upcoming requests [44]. In short, executing request updating in an online manner is not practical in real-production environment. Due to the above facts, current works mainly focus on online scheduling [45], [46], [47] and

TABLE II

ALGORITHM COMPARISON. ERP WILL UPGRADE THE CONFIGURATION OF NF_1 AND NF_2 TO TYPE B, WHICH COSTS \$491 AND NEGLIGIBLE UPDATE DELAY. RU NEEDS TO UPDATE 1500/600 REQUESTS FROM NF_1/NF_2 TO NF_3, WHICH COSTS 21 UNITS UPDATE DELAY AND \$291. TRUST WILL TRANSFER 600 REQUESTS FROM NF_2 TO NF_3, AND UPGRADE THE CONFIGURATION TYPE OF NF_1. THIS COSTS \$391 AND 6 UNITS UPDATE DELAY

Method	NF Type			NF Traffic (Gbps)			Money	Update Delay
	NF_1	NF_2	NF_3	NF_1	NF_2	NF_3		
Original	A	A	A	65	56	27	\$291	\
ERP	**B↑**	**B↑**	A	65	56	27	\$491	≈ 0
RU	A	A	A	**50↓**	**50↓**	**48↑**	\$291	21
TRUST	**B↑**	A	A	65	**50↓**	**33↑**	\$391	6

offline updating [48], [49], [50] for requests. Like the existing works, we choose to schedule the upcoming requests in an online manner and offline update requests.

Meanwhile, to ensure that the decisions over time will not oscillate, we will set load ratio threshold for updating. Specifically, according to [28], the configuration of an NF can be upgraded when the load ratio is greater than threshold A (*e.g.*, 80%). When executing the updating algorithm, we can limit the load ratio below threshold B (*e.g.*, 70%). Meanwhile, we can set threshold C of load ratio to downgrade the configuration (*e.g.*, 50%). Thus, when the load is between thresholds A and C, the updating algorithm will not be triggered. In all, according to network state, setting thresholds reasonably can effectively avoid decisions oscillating.

In the next section, we will give the problem formulation of joint elastic resource provisioning and request updating, *i.e.*, TRUST. Correspondingly, we show the NP-Hardness of the problem and propose an approximate algorithm with bounded approximation factors.

D. Problem Formulation

This section describes the main problem formulation of TRUST. In a commercial cloud, the service provider serves the requests from cloud users and processes the requests in NFs. As time goes by, load unbalancing occurs among NFs due to traffic dynamics. At this time, the service provider will choose to change the configuration type of some NFs or update the requests. To help the service provider spend less money on NFs and save time from updating requests, also for achieving a better QoS of users, we should consider the following constraints in coping with traffic dynamics.

1) *Physical Server Resource Constraint:* The total resources used by NFs that are on the same physical server cannot exceed the whole resources of the physical server.

2) *NF Capacity Constraint:* The configuration type that an NF chooses must be able to handle the requests it receives.

3) *Update Delay Constraint:* t_0 represents the update delay of a single request, which is determined by the system hardware performance. Considering various factors, it is difficult to determine a representative value for controller update delay as we discussed in the Introduction. However, we should note that, no matter which mode is adopted or what topology is deployed, the processing speed of the controller is always restricted. Meanwhile, the update delay t_0 defined in our problem formulation is easy to be modified when facing different situations. In this paper, we generally consider different main-stream controllers like ONOS and OpenDayLight and choose the medium number 0.5ms as a case study. Since the controller will encapsulate and install a flow entry for each updated request, the total update delay can be approximately linear with the number of updated requests [44], [51]. To assure users' QoS, the total update delay cannot exceed the time threshold T, which is determined by the user's QoS demand. For instance, in a cellular communication environment, T is set to 150ms [34].

4) *Objective Function:* Our object is to minimize the infrastructure cost that the service provider needs to pay when satisfying those constraints above.

We use the variable $x_c^n \in \{0,1\}$ to denote whether NF n will choose the configuration type c ($x_c^n = 1$) or not ($x_c^n = 0$). The variable $y_n^\gamma \in \{0,1\}$ represents whether the request γ will be sent to NF n ($y_n^\gamma = 1$) or not ($y_n^\gamma = 0$). We use a constant $\beta(\gamma, n)$ to denote whether the request γ is assigned to NF n before update ($\beta(\gamma,n) = 0$) or not ($\beta(\gamma,n) = 1$).

The formulation is as follows:

$$\min \sum_{n\in N}\sum_{c\in C} m(c)\cdot x_c^n$$
$$S.t. \begin{cases} \sum_{c\in C} x_c^n = 1, & \forall n \in N \\ \sum_{n\in N} y_n^\gamma = 1, & \forall \gamma \in \Gamma \\ \sum_{n\in N_s}\sum_{c\in C} x_c^n \cdot r(c) \le R(s), & \forall s \in S \\ \sum_{\gamma\in\Gamma} y_n^\gamma \cdot f(\gamma) \le \sum_{c\in C} x_c^n \cdot p(c), & \forall n \in N \\ \sum_{\gamma\in\Gamma}\sum_{n\in N} y_n^\gamma \cdot \beta(\gamma,n)\cdot t_0 \le T, & \\ x_c^n, y_n^\gamma \in \{0,1\}, & \forall n, c, \gamma \end{cases}$$

The first set of equations means that each NF will choose a configuration type $c \in C$. We should note that, the first set of equations, *i.e.*, $\sum_{c\in C} x_c^n = 1,\ \forall n \in N$ means that every NF will choose an appropriate configuration according to current load, *i.e.*, upgrading the configuration when facing overload or downgrading the configuration when facing underload. The second set of equations means each request γ will be sent to an NF $n \in N$. The third set of inequalities denotes the physical server resource constraint that all the resources allocated to NFs should not exceed the total resources on the physical server s. The fourth set of inequalities describes that the traffic load on each NF should not exceed its capacity, *i.e.*, the NF capacity constraint. The fifth set of inequalities means that the duration of updating request should not exceed the time threshold T. Our objective function is to minimize the total infrastructure cost, *i.e.*, $\sum_{n\in N}\sum_{c\in C} m(c)\cdot x_c^n$.

Theorem 1: TRUST defined in Eq. (1) is an NP-Hard problem.

Proof: We can prove the NP-hardness by showing that the bin-packing problem [52] is a special case of TRUST. Specifically, we first assume all the NFs are overloaded. Then, for each NF, we divide the requests into two parts: the first part of requests are within the NF's capacity and the second part of requests are those beyond NF's capacity. So we have to upgrade each NF to a configuration type to hold the second part of requests. This equals we need to find extra NFs with appropriate configurations to handle them. Finally, we assume there exists only one configuration type for NFs. This equals how to pack these requests into knapsacks while achieving the minimum costs, *i.e.*, the bin-packing problem. This concludes the NP-Hardness of our problem. □

E. Problem Complexity Analysis

In fact, the problem remains challenging if we simplify our problem by ignoring the variables associated with elastic resource provisioning, *i.e.*, x_c^n. The simplified problem can be formulated as follows:

$$\min\ h(L_n)$$
$$S.t.\ \begin{cases} \sum_{n\in N} y_n^\gamma = 1, & \forall \gamma\in\Gamma \\ \sum_{\gamma\in\Gamma}\sum_{n\in N} y_n^\gamma \cdot \beta(\gamma,n)\cdot t_{\gamma,n} \le T, & \\ L_n = \sum_{\gamma\in\Gamma} y_n^\gamma \cdot f(\gamma), & \forall n\in N \\ y_n^\gamma \in\{0,1\}, & \forall n,\gamma \end{cases}$$

We define a function h to calculate the infrastructure cost of NFs. The input of the function is the load of NF n, *i.e.*, L_n. When h is convex, our objective is to minimize L_n. Otherwise, we need to maximize L_n. The first constraint means every request needs to be scheduled to an NF. The second constraint represents the update delay threshold. The third constraint calculates the load on every NF. All the variables are binary integers. The simplified problem is a special case of the DSRU (Delay-Satisfied Route Update) problem defined in [44]. The authors in [44] proved the NP-Hardness by showing multi-commodity flow with minimum congestion problem [53] is a special case of the DSRU problem and proposed a rounding-based algorithm to solve it. It shows the complexity of the kind of problems and also implies that currently there may not exist any more efficient algorithms to solve it. Meanwhile, our problem is much more difficult than DSRU problem, since we have another variable to determine the configuration for each NF, *i.e.*, x_c^n. In our algorithm design, we will show how to achieve a feasible solution with bounded approximation factors.

F. Discussions

1) Update Delay: In this paper, we mainly consider the situation where the updating of requests is executed in the same datacenter, and the delay is simulated as a constant t_0 [54], [55]. However, we further show that our algorithm can still handle the case where geographically distributed datacenters exist. Specifically, we can change the constant t_0 into $t_{\gamma,n}$, denoting the delay of transferring request γ to the NF n. If the request does not need to be updated, *i.e.*, the NF remains the same, the update delay $t_{\gamma,n} = 0$. The modified formulation is as follows:

$$\min \sum_{n\in N}\sum_{c\in C} m(c)\cdot x_c^n$$
$$S.t.\ \begin{cases} \sum_{c\in C} x_c^n = 1, & \forall n\in N \\ \sum_{n\in N} y_n^\gamma = 1, & \forall\gamma\in\Gamma \\ \sum_{n\in N_s}\sum_{c\in C} x_c^n\cdot r(c) \le R(s), & \forall s\in S \\ \sum_{\gamma\in\Gamma} y_n^\gamma\cdot f(\gamma) \le \sum_{c\in C} x_c^n \cdot p(c), & \forall n\in N \\ \sum_{\gamma\in\Gamma}\sum_{n\in N} y_n^\gamma\cdot\beta(\gamma,n)\cdot t_{\gamma,n}\le T, & \\ x_c^n, y_n^\gamma \in\{0,1\}, & \forall n,c,\gamma \end{cases}$$

Since $t_{\gamma,n}$ is only determined by the request and the NF, it is also a constant, which means the feasibility of our algorithm. If the physical distance between a request and a datacenter is too long, the value of $t_{\gamma,n}$ would be very large. That is to say, our algorithm is unlikely to acquire the corresponding transfer solution. We can similarly use the proposed algorithm to derive a feasible solution with bounded approximation factors.

2) NF Configuration: Practically, cloud vendors provide both fixed and configurable instances for clients. Specifically, a fixed instance is provisioned in a constant capacity, *i.e.*, each configuration is associated with a certain number of CPU cores, memory, storage, etc. And a configurable instance allows clients to choose resource in a more fine-grained manner. Compared with the fixed type, a configurable instance can provide resource in a more fine-grained manner. However, it also means a higher capability of resource management and more complex resource provisioning [28], which is why current mainstream cloud vendors, like Amazon, Google and Microsoft all prefer the fixed configuration. Thus, in this paper, we mainly focus on the fixed configuration rather than configurable instances.

3) Elasticity Scaling: There are two kinds of scaling in elastic resource provisioning. Specifically, the first one is called horizontal scaling, which directly adds or removes instances. The second is called vertical scaling, which increases or decrease network resources of instances, like CPU cores, memory or network bandwidth [28]. Currently in both academia and industry, both horizontal scaling and vertical scaling are adopted. We mainly focus on vertical scaling in this paper, because of the following advantages [56], [57], [58]. First, vertical scaling is more fine-grained, since it allows to add or remove units of resources. Second, vertical scaling can eliminate the overhead caused by booting extra instances, and it does not need additional machines like load balancers. Finally, it guarantees the sessions of requests will not be interrupted when scaling. That is to say, vertical scaling can guarantee users' QoS. Therefore, as a prospective study, our work focuses on the promising situation where vertical scaling is adopted.

III. Algorithm Description

A. Randomized Rounding Algorithm for TRUST

This section presents an approximation algorithm based on randomized rounding [59] for TRUST. The reason for choosing the randomized rounding algorithm mainly includes the following aspects. First, it is an efficient approximate algorithm that can achieve a feasible solution close to the optimal value with a high probability. Second, it will not exceed the constraint by a bounded factor. Overall, it is applicable and efficient to design an approximate algorithm based on the randomized rounding algorithm.

In specific, the first step is to relax Eq. (1) into an LP by replacing integer constraints with linear constraints. In this way, we can solve it with a linear program solver (*e.g.*, PuLP [60]) and the solutions are denoted by $\{\widetilde{x}_c^n\}$ and $\{\widetilde{y}_n^\gamma\}$.

In the second step, we identify which configuration type each NF should choose and which NF each request should be scheduled to, *i.e.*, obtain feasible solutions $\{\widehat{x}_c^n\}$ and $\{\widehat{y}_n^\gamma\}$. Then, we round $\widehat{x}_c^n$ to 1 with probability $\widetilde{x}_c^n$ and round $\widehat{y}_n^\gamma$ to 1 with probability $\widetilde{y}_n^\gamma$. We should note that the choice is done in an exclusive manner, *i.e.*, for each n, exactly one of $\{\widehat{x}_c^n\}$ is set to one; the rest are set to zero. And for each γ, exactly one of $\{\widehat{y}_n^\gamma\}$ is set to one; the rest are set to zero. After we acquire the final results of algorithm, i.e., $\{\widehat{x}_c^n\}$ and $\{\widehat{y}_n^\gamma\}$, we can determine which configuration an NF adopts and which NF a request is assigned to. If $\widehat{x}_c^n = 1$, NF n chooses the configuration c, and if $\widehat{y}_n^\gamma = 1$, request γ is assigned to the NF n. The formal algorithm is shown in Algorithm 1.

As discussed in Section II-C, Algorithm 1 will be triggered by a new-arrival request or a timer. Thus, how to set a proper timer is also an important issue. If we execute Algorithm 1 at a long time interval, the network performance may become worse due to traffic dynamics. However, if we execute Algorithm 1 too frequently, it may affect too many requests in the cloud, decreasing the whole cloud network performance. Usually, we determine the interval of executing Algorithm 1 according to the cloud network status (*e.g.*, 50 minutes). To further promote network performance during the interval, we propose the improved scheme TRUST(x%), where x represents the percentage of selected requests and NFs. Specifically, we set a short time interval (*e.g.*, 10 minutes) and select a small part (*e.g.*, 1%) of overloaded NFs and corresponding requests to execute Algorithm 1 for updating. Then, Algorithm 1 will be executed at the 10th, 20th, 30th, 40th minutes to **partially** update selected requests and NFs, and at the 50th minutes, we will run Algorithm 1 to **globally** update requests and NFs. In a word, TRUST and TRUST(x%) are two different schemes and they are executed parallelly and separately. TRUST(x%) can be regarded as a modified and improved version of TRUST. The value of x must be small to avoid affecting cloud network performance significantly. Otherwise, the involved requests and NFs may be too many, decreasing the network performance. In this paper, we mainly consider when $x = 1, 2, 5$, how much can TRUST(x%) improve the performance. The performance comparison between TRUST and TRUST(x%) will be assessed through our simulations in Section IV-B.2.

Algorithm 1 Randomized Rounding-Based Algorithm for TRUST

```
1:  Step 1: Solving the Relaxed Formulation
2:  Construct a linear program by replacing the integral constraints with $x_c^n$ and $y_n^\gamma \in [0,1]$
3:  Obtain the optimal solutions $\{\widetilde{x}_c^n\}$ and $\{\widetilde{y}_n^\gamma\}$
4:  Step 2: Acquire a feasible solution by randomized rounding
5:  for each NF $n \in N$ do
6:      Choose one $\widehat{x}_c^n = 1$ with probability $\widetilde{x}_c^n$ and rest are set 0
7:  end for
8:  for each request $\gamma \in \Gamma$ do
9:      Choose one $\widehat{y}_n^\gamma = 1$ with probability $\widetilde{y}_n^\gamma$ and rest are set 0
10: end for
11: Step 3: Derive a joint elastic resource provisioning and request updating scheme according to $\{\widehat{x}_c^n\}$ and $\{\widehat{y}_n^\gamma\}$
```

B. Performance Analysis

Before we analyze algorithm performance, we introduce two famous lemmas for approximation performance analysis.

Lemma 2 (Chernoff Bound): Given n independent variables: $x_1, x_2, \ldots, x_n$, where $\forall x_i \in [0,1]$. Let $\mu = \mathbb{E}[\sum_{i=1}^n x_i]$. Then, we have $\mathbf{Pr}[\sum_{i=1}^n x_i \geq (1+\epsilon)\mu] \leq e^{\frac{-\epsilon^2\mu}{2+\epsilon}}$ and $\mathbf{Pr}[\sum_{i=1}^n x_i \leq (1-\epsilon)\mu] \leq e^{\frac{-\epsilon^2\mu}{2}}$ where ϵ is an arbitrarily positive value.

Lemma 3 (Union Bound): Given an accountable set of n events: $A_1, A_2, \ldots A_n$, each event A_i happens with probability $\mathbf{Pr}(A_i)$. Then, $\mathbf{Pr}(A_1 \cup A_2 \cup \ldots \cup A_n) \leq \sum_{i=1}^n \mathbf{Pr}(A_i)$.

Theorem 4: The proposed algorithm can obtain a value of infrastructure cost close to the optimal infrastructure cost derived by solving the LP with a high probability.

Proof: We define the infrastructure cost of NF n: $P_n = \sum_{c\in C} x_c^n \cdot m(c)$. Then, the total infrastructure cost, *i.e.*, the objective function can be denoted as: $\sum_{n\in N} P_n$. According to Algorithm 1, we know the infrastructure cost of NF n derived by LP, *i.e.*, the optimal value is $\sum_{n\in N} \widetilde{P}_n = \sum_{n\in N}\sum_{c\in C} \widetilde{x}_c^n \cdot m(c)$. After the rounding procedure of algorithm, we acquire the feasible solution $\widehat{x}_c^n$. Thus, we can determine the infrastructure cost derived by algorithm, *i.e.*, $\widehat{P}_n$:

$$\widehat{P}_n = \begin{cases} m(c), & \textit{with probability } \widetilde{x}_c^n \\ 0, & \textit{otherwise} \end{cases} \tag{1}$$

So we have $\mathbb{E}[\sum_{n\in N} \widehat{P}_n] = \sum_{n\in N}\sum_{c\in C} \widetilde{x}_c^n \cdot m(c) = \sum_{n\in N} \widetilde{P}_n$. Then, we define the highest price of configuration type is p_{max}. So we can say $\mathbb{E}[\frac{\sum_{n\in N} \widehat{P}_n}{|N|\cdot p_{max}}] \in [0,1]$, where $|N|$ is the number of NFs. We should note that for different n, $\widehat{P}_n$ is independent from each other. Then, based on Lemma 2, we can conclude that

$$\mathbf{Pr}[\frac{\sum_{n\in N} \widehat{P}_n}{|N|\cdot p_{max}} \geq (1+\epsilon)\mathbb{E}[\frac{\sum_{n\in N} \widehat{P}_n}{|N|\cdot p_{max}}]] \leq e^{-\frac{\epsilon^2}{2+\epsilon}\mathbb{E}[\frac{\sum_{n\in N} \widehat{P}_n}{|N|\cdot p_{max}}]}$$
$$\Leftrightarrow \mathbf{Pr}[\sum_{n\in N} \widehat{P}_n \geq (1+\epsilon)\sum_{n\in N} \widetilde{P}_n] \leq e^{-\frac{\epsilon^2}{2+\epsilon}\frac{\sum_{n\in N} \widetilde{P}_n}{|N|\cdot p_{max}}} \tag{2}$$

Eq. (2) means that the infrastructure cost derived by algorithm, *i.e.*, $\sum_{n\in N} \widehat{P}_n$ will be more than the value derived by LP, *i.e.*, $\sum_{n\in N} \widetilde{P}_n$ with a very low probability. Thus, we can say the output derived by algorithm, *i.e.*, $\sum_{n\in N} \widehat{P}_n$ is close to the optimal solution of LP, *i.e.*, $\sum_{n\in N} \widetilde{P}_n$ with a high probability. □

Theorem 5: The algorithm guarantees that the total resources of NFs in server s will not exceed the total resource of server s by a factor of $O(\log|S|)$, where $|S|$ is the number of NFs.

Proof: We use $\mathbb{R}_n$ to denote the resource of NF n and $\mathbb{R}_n = \sum_{c\in C} x_c^n \cdot r(c)$. After solving the LP, $\widetilde{\mathbb{R}}_n = \sum_{c\in C} \widetilde{x}_c^n \cdot r(c)$. Then, we adopt the rounding procedure of $\widetilde{x}_c^n$ to acquire a feasible solution $\widehat{x}_c^n \in \{0,1\}$ and $\widehat{\mathbb{R}}_n$. Specifically, we round $\widehat{x}_c^n$ to 1 with probability $\widetilde{x}_c^n$. Thus, we can determine the capacity of NF n derived by algorithm, *i.e.*, $\widehat{\mathbb{R}}_n$:

$$\widehat{\mathbb{R}}_n = \begin{cases} r(c), & \text{with probability } \widetilde{x}_c^n \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

So we have

$$\mathbb{E}[\widehat{\mathbb{R}}_n] = \sum_{c\in C} \widetilde{x}_c^n \cdot r(c) = \widetilde{\mathbb{R}}_n \tag{4}$$

and

$$\mathbb{E}[\sum_{n\in N_s} \widehat{\mathbb{R}}_n] = \sum_{n\in N_s}\sum_{c\in C} \widetilde{x}_c^n \cdot r(c) \le R(s) \tag{5}$$

Then we define a constant α as follows:

$$\alpha = \frac{R_{min}}{r_{max}} \tag{6}$$

where R_{min} denotes the minimal resource among servers and r_{max} denotes the maximum resource among configuration types.

Combining the definition of α and Eq. (5), we have

$$\begin{cases} \dfrac{\widehat{\mathbb{R}}_n \cdot \alpha}{R(s)} \in [0,1] \\ \mathbb{E}[\displaystyle\sum_{n\in N_s} \frac{\widehat{\mathbb{R}}_n \cdot \alpha}{R(s)}] \le \alpha \end{cases} \tag{7}$$

We should note that for different n, $\widehat{\mathbb{R}}_n$ is independent from each other. Based on Lemma 2, we have

$$\mathbf{Pr}[\sum_{n\in N_s} \frac{\widehat{\mathbb{R}}_n \cdot \alpha}{R(s)} \ge (1+\epsilon)\alpha] \le e^{-\frac{\epsilon^2\cdot\alpha}{2+\epsilon}} \tag{8}$$

Now we assume that

$$\mathbf{Pr}[\sum_{n\in N_s} \frac{\widehat{\mathbb{R}}_n}{R(s)} \ge (1+\epsilon)] \le e^{-\frac{\epsilon^2\cdot\alpha}{2+\epsilon}} \le \frac{1}{|S|^{k+1}} \tag{9}$$

where k is an arbitrary positive integer. We know that $\frac{1}{|S|^{k+1}} \to 0$ when network size grows. The solution to Eq. (9) is

$$\epsilon \ge \frac{(k+1)\log|S| + \sqrt{(k+1)^2\log^2|S| + 8(k+1)\alpha\log|S|}}{2\alpha}, \tag{10}$$

$$\Rightarrow \epsilon \ge \frac{(k+1)\log|S|}{\alpha} + 2, \quad |S| \ge 2 \tag{11}$$

By applying Lemma 3, we have

$$\begin{aligned} &\mathbf{Pr}[\bigcup_{s\in S}\sum_{n\in N_s} \frac{\widehat{\mathbb{R}}_n}{R(s)} \ge (1+\epsilon)] \\ &\le \sum_{s\in S} \mathbf{Pr}[\sum_{n\in N_s} \frac{\widehat{\mathbb{R}}_n}{R(s)} \ge (1+\epsilon)] \\ &\le |S| \cdot \frac{1}{|S|^{k+1}} = \frac{1}{|S|^k} \end{aligned} \tag{12}$$

Thus, we can conclude that the approximation factor is $\epsilon + 1 = \frac{(k+1)\log|S|}{\alpha} + 3$. □

Lemma 6: Suppose that $\widetilde{C}_n$ is the optimal value of the capacity of NF n to the LP while $\widehat{C}_n$ is the value associated with Algorithm 1. $\widehat{C}_n$ will be at most less than $\widetilde{C}_n$ by a factor of $(1-\epsilon)$, where ϵ is an arbitrarily positive value.

Proof: We use C_n to denote the capacity of NF n and $C_n = \sum_{c\in C} x_c^n \cdot p(c)$. After solving the LP, $\widetilde{C}_n = \sum_{c\in C} \widetilde{x}_c^n \cdot p(c)$. Then, we adopt the rounding procedure of $\widetilde{x}_c^n$ to acquire a feasible solution $\widehat{x}_c^n \in \{0,1\}$. Specifically, we round $\widehat{x}_c^n$ to 1 with probability $\widetilde{x}_c^n$. Thus, we can determine that

$$\widehat{C}_n = \begin{cases} p(c), & \text{with probability } \widetilde{x}_c^n \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

So we have

$$\mathbb{E}[\widehat{C}_n] = \sum_{c\in C} \widetilde{x}_c^n \cdot p(c) = \widetilde{C}_n \tag{14}$$

Clearly, we also have $\frac{\widehat{C}_n}{c_{max}}$ and $\mathbb{E}[\frac{\widehat{C}_n}{c_{max}}] \in [0,1]$. Specifically, c_{max} represents the maximum capacity among all the NFs and c_{min} represents the minimum capacity among all the NFs. For different n, $\widehat{C}_n$ is independent from each other. Based on Lemma 2, we have

$$\mathbf{Pr}[\frac{\widehat{C}_n}{c_{max}} \le (1-\epsilon)\mathbb{E}[\frac{\widehat{C}_n}{c_{max}}]] \le e^{-\frac{\epsilon^2}{2}\mathbb{E}[\frac{\widehat{C}_n}{c_{max}}]} \tag{15}$$

Combining Eq. (14), then we have

$$\mathbf{Pr}[\widehat{C}_n \le (1-\epsilon)\widetilde{C}_n] \le e^{-\frac{\epsilon^2}{2}\frac{\widetilde{C}_n}{c_{max}}} \tag{16}$$

Thus, we can conclude that after rounding procedure, Algorithm 1 can derive a throughput, *i.e.*, $\widehat{C}_n$, will not be less than $(1-\epsilon)\widetilde{C}_n$. □

Theorem 7: Algorithm 1 can achieve the approximation factor of $O(\log|N|)$ for NF capacity, where $|N|$ is the number NFs.

Proof: By Lemma 6, we can know that Algorithm 1 will acquire a value of NF capacity $\widehat{C}_n$ will not be less than $(1-\epsilon)\widetilde{C}_n$. Then, we will show that the total traffic load on each NF will not exceed $\widehat{C}_n$ by a factor of $O(\log|N|)$.

We use a variable v_n^r to denote the traffic size of request r to NF n.

$$v_n^r = \begin{cases} f(r), & \text{with probability } \widetilde{y}_n^r \\ 0, & \text{otherwise} \end{cases} \tag{17}$$

So we have

$$\mathbb{E}[\sum_{r\in\Gamma} v_n^r] = \sum_{r\in\Gamma} f(r)\cdot \widetilde{y}_n^r \le \sum_{c\in C} \widetilde{x}_c^n \cdot p(c) = \widetilde{C}_n \tag{18}$$

We define:

$$\alpha_2 = \min\{\frac{c_{min}}{\widetilde{y}_n^r \cdot f(r)_{max}}\} \tag{19}$$

Combining Eq. (18) and Eq. (19), we have

$$\begin{cases} \dfrac{v_n^r \cdot \alpha_2}{\widetilde{C}_n} \in [0,1] \\ \mathbb{E}[\displaystyle\sum_{r\in\Gamma} \frac{v_n^r \cdot \alpha_2}{\widetilde{C}_n}] \le \alpha_2 \end{cases} \tag{20}$$

Note that when n is given, for different r, v_n^r is independent from each other. Thus, by adopting Lemma 2, we have

$$\mathbf{Pr}[\sum_{r\in\Gamma} \frac{v_n^r \alpha_2}{\widetilde{C}_n} \geq (1+\epsilon')\alpha_2] \leq e^{\frac{-\epsilon^2\cdot\alpha_2}{2+\epsilon}} \tag{21}$$

Now we assume that

$$\mathbf{Pr}[\sum_{r\in\Gamma} \frac{v_n^r}{\widetilde{C}_n} \geq (1+\epsilon')] \leq e^{\frac{-\epsilon^2\cdot\alpha_2}{2+\epsilon}} \leq \frac{1}{|N|^2} \tag{22}$$

We know that $\frac{1}{|N|^2} \rightarrow 0$ when the network grows. By solving Eq. (22), we have the following result:

$$\epsilon' \geq \frac{\log|N|^2 + \sqrt{\log^2|N|^2 + 8\alpha_2\log|N|^2}}{2\alpha_2}, \quad n \geq 2$$
$$\Rightarrow \epsilon' \geq \frac{2\log|N|}{\alpha_2} + 2, \quad n \geq 2 \tag{23}$$

Then, by applying Lemma 3, we have

$$\begin{aligned}&\mathbf{Pr}[\bigcup_{n\in N}\sum_{r\in\Gamma}\frac{v_n^r}{\widetilde{C}_n} \geq (1+\epsilon')]\\ &\leq \sum_{n\in N}\mathbf{Pr}[\sum_{r\in\Gamma}\frac{v_n^r}{\widetilde{C}_n} \geq (1+\epsilon')]\\ &\leq |N|\cdot\frac{1}{|N|^2} = \frac{1}{|N|}\end{aligned} \tag{24}$$

Combining Lemma 6, we can conclude that the approximation factor is

$$\frac{1+\epsilon'}{1-\epsilon} = \frac{1}{1-\epsilon}(\frac{2\log|N|}{\alpha_2}+3), \tag{25}$$

where ϵ is an arbitrarily positive value. □

Theorem 8: Our algorithm can guarantee that the update time derived by Algorithm 1 will not exceed the time threshold T by a factor of $O(\log|\Gamma|)$, where $|\Gamma|$ denotes the number of requests.

Proof: The proof of Theorem 8 is similar to the proof of Theorem 5. Due to limited space, we omit the proof. □

Approximation Factor: From our analysis, we can make the following conclusions. First of all, the infrastructure cost derived by algorithm is close to the optimal value derived by solving the LP with a high probability. Secondly, the total capacity of the NFs on the same physical server will not exceed the total resource of the server by a factor of $O(\log|S|)$, where $|S|$ means the number of servers. Thirdly, the total request load on each NF will not exceed the capacity of the NF by a factor of $O(\log|N|)$, where $|N|$ is the number of NFs. Finally, the time threshold for updateing will hardly be violated by a factor of $O(\log|\Gamma|)$, where $|\Gamma|$ denotes the number of requests.

IV. Performance Evaluation

A. Performance Metrics and Benchmarks

Performance Metrics: We adopt the following performance metrics in evaluations: (1) the infrastructure cost; (2) the system throughput; (3) the badput ratio [61]; (4) the update delay; (5) the packet loss ratio; (6) the Round-Trip Time (RTT) and (7) the flow completion time (FCT).

During a simulation run, we record each NF's configuration type and calculate the total infrastructure cost that the service provider needs to pay. Then, we measure the total load of all the NFs as the throughput from users that the service provider can serve. Note that not all the traffic from users can be served due to limited capacity. We define the amount of traffic which cannot be served by NFs due to overload or network congestion as badput. Then, we divide badput by the total traffic amount from users as the badput ratio [61]. It is a key metric to quantify the QoS and network performance of a cloud network. Finally, we calculate the update delay according to the number of rules that the controller needs to generate. During a system implementation run, we measure the packet loss ratio and the FCT using the command iPerf3.

Benchmarks: We compare TRUST with three state-of-the-art benchmarks dealing with traffic dynamics in clouds. The first benchmark is the Elastic Resource Provisioning Reactive Mode (ERP-RM) algorithm [28], which is widely adopted in commercial clouds like Amazon, Scalr and Rightscale. ERP-RM sets the capacity threshold for each NF and automatically selects the configuration type with sufficient resources. The second benchmark is the Robustness-aware Real-time Request Updating Algorithm (R^3-UA) [37]. R^3-UA adopts the rounding method to acquire the real-time request updating scheme in order to achieve load-balancing among NFs. This benchmark also has update delay assurance by limiting the number of updating requests. The third benchmark is the Request Updating-Shortest Job First (RU-SJF) algorithm [38], which is highly efficient and also widely adopted in clouds. RU-SJF always chooses the NF with the least burden for the request with the least traffic demand. R^3-UA and RU-SJF are both pure request updating methods and do not involve modifying configuration types. Note that R^3-UA considers the update delay constraint, while RU-SJF does not. We should note that, since existing works only adopt the elastic resource provisioning or pure request updating to handle traffic dynamics, we choose the state-of-the art algorithms in each aspect as our benchmarks.

B. Simulation Evaluation

1) Simulation Settings: We choose both small- and large-scale datacenter from Google Cluster Data [62]. The small-scale contains 16 NFs and the large-scale datacenter contains 320 NFs. Each request includes different properties, like start/end time of the measurement period, job ID, NF ID, request size, etc. We mainly leverage the NF ID and request size to simulate distributing requests. The dynamic situations are simulated according to [63]. The requests are divided into two parts: primitive requests and newly increased requests. Specifically, there are mainly two kinds of traffic dynamics. One is called slight dynamic, where newly increased requests account for 20% of all the requests. The other is called magnitude dynamic, where newly increased requests account for 50% of all the requests. During both two kinds of traffic dynamics, 20% of NFs will receive newly increased requests and 20% of NFs will reduce about 50% of primitive requests. The configuration type is set according to Google Cloud Platform [29]. A physical server can provide 100 core CPU and 375 GB RAM for NFs. The configuration types are shown in Table III.

We conduct simulation experiments under the two kinds of traffic dynamics. Besides, we generate $6\times10^3/6\times10^4$ requests for the small/large topology by default. The update delay constraint is set to 0.2/2s for the small/large topology by default.

TABLE III
CONFIGURATION TYPES ACCORDING TO GOOGLE CLOUD PLATFORM [29]

Type	Resource		Price	Capacity
	CPU (cores)	RAM (G)	($/hour)	(Gbps)
1	1	3.75	0.07	1
2	2	7.5	0.14	2
3	4	15	0.28	4
4	8	30	0.57	8
5	16	60	1.13	16
6	32	120	2.26	32
7	64	240	3.40	64

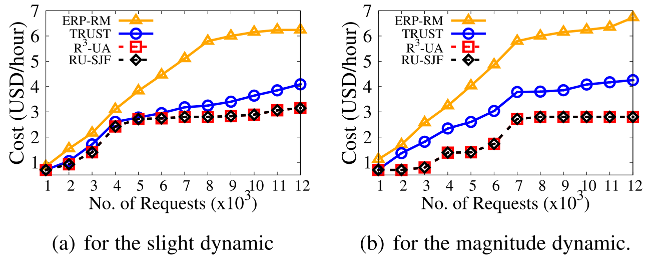

(a) for the slight dynamic (b) for the magnitude dynamic.

Fig. 2. Cost vs number of requests in the small topology.

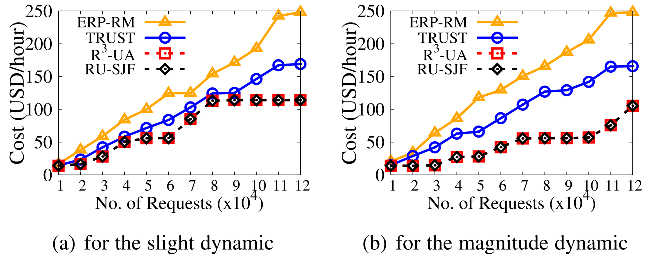

(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 3. Cost vs number of requests in the large topology.

2) Performance Evaluation: We run three groups of simulation experiments to check the effectiveness of our algorithm. The results are shown in Figs. 2-17. Specifically, the first group of simulation experiments shows the infrastructure cost evaluations. The second group of experiments shows the QoS-related metrics (*e.g.*, throughput, badput ratio and update delay) evaluations. The third group of experiments shows the performance comparison between TRUST and TRUST($x\%$).

In the first group of experiments, we mainly focus on the infrastructure cost results. Figs. 2-3 show the infrastructure costs by varying the number of requests in clouds. We can learn from the figures that the costs of four algorithms increase when the number of requests grows. The figures show that our proposed algorithm always acquires a much lower cost than ERP-RM and a slightly higher cost than R^3-UA and RU-SJF. For example, in Fig. 2(a), given 8×10^3 requests in the cloud, the cost results of four algorithms are 6, 3.4, 2.83 and 2.83 USD/hour, corresponding to ERP-RM, TRUST, R^3-UA and RU-SJF, respectively. TRUST reduces the cost by 43.96% compared with ERP-RM while only increasing the cost by 13.79% compared with both R^3-UA and RU-SJF. Since R^3-UA and RU-SJF will not buy any more extra resources when facing traffic dynamics, it is natural that the cost results will be lower than those of ERP-RM and TRUST. As a result, the QoS aspects (*e.g.*, throughput and update delay) will decrease significantly, which will be clarified hereafter. In Fig. 3(a), when there are 9×10^4 requests, the cost results of four algorithms are 171.42, 125.1, 114 and 114 USD/hour, corresponding to ERP-RM, TRUST, R^3-UA and RU-SJF. TRUST reduces the cost by 27% compared with ERP-RM.

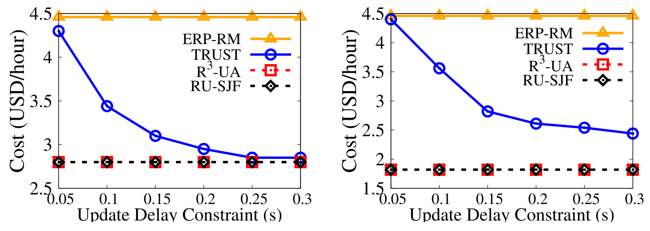

(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 4. Cost vs. Update delay constraint in the small topology.

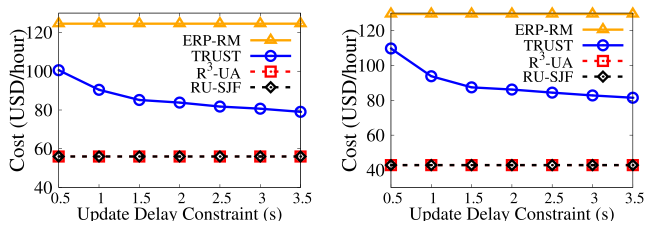

(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 5. Cost vs. Update delay constraint in the large topology.

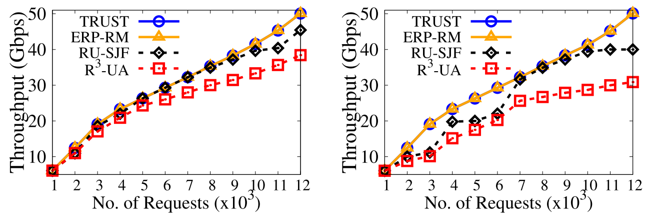
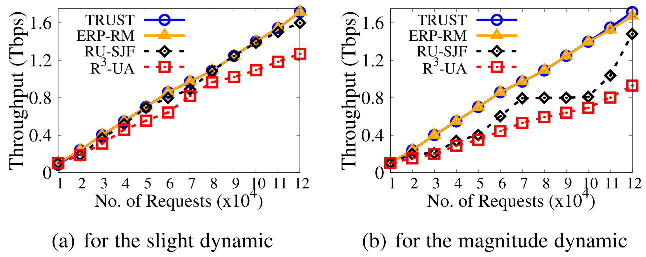

(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 6. Throughput vs. Number of requests in the small topology.

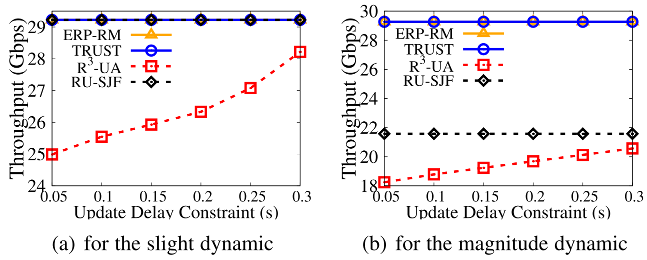
Figs. 4-5. show the infrastructure costs by varying the update delay constraint and the results are shown in It can be concluded from the figures that, given a larger update delay threshold, TRUST will acquire a solution with less cost. Also, since ERP-RM only executes elastic resource provisioning, and R^3-UA/RU-SJF only executes updating requests, their cost results are steady. For instance, in Fig. 4(a), the cost results of ERP-RM, R^3-UA and RU-SJF are 4.46, 2.8 and 2.8 USD/hour, respectively. Obviously, the cost results of the three benchmarks will not be affected by the update delay constraint. When the update delay constraint is set as 0.05s, TRUST is only able to transfer about tens of requests. The cost result of TRUST is nearly the same as that of ERP-RM. However, when given sufficient time, like 0.2s, the cost result of TRUST is only 2.95 USD/hour. TRUST reduces the cost by 33.9% compared with ERP-RM and increases the cost only by about 5.4% compared with both R^3-UA and RU-SJF. The figures also imply that when facing the magnitude dynamics, pure request updating methods will not be able to solve the ultimate problem, *i.e.*, insufficient resources to deal with the sudden increasing number of requests. For instance, in Fig. 5(b), even if TRUST has enough time to update requests, the cost is still a bit higher than those of R^3-UA and RU-SJF. This is because extra resources are necessary, and the consequence of not upgrading the configuration type is to decline redundant requests, resulting in a bad users' QoS.

In the second group of experiments, we mainly focus on QoS-related matrics (*e.g.*, throughput, badput ratio and update delay). Figs. 6-7 show the throughput results by varying



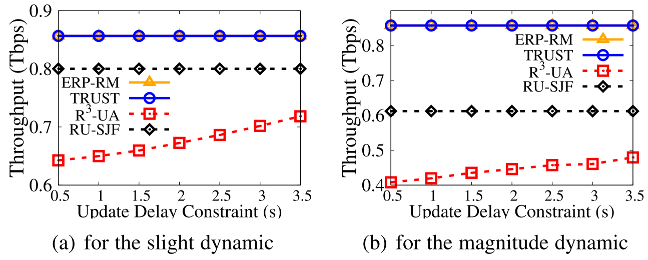
(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 7. Throughput vs. Number of requests in the large topology.



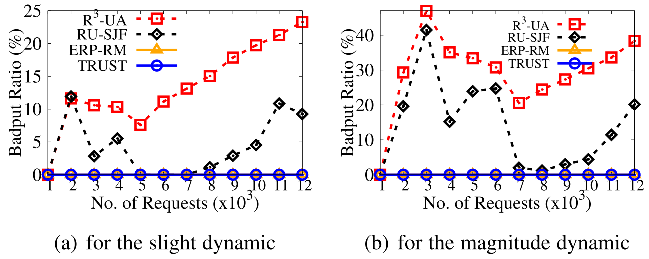
(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 8. Throughput vs. Number of requests in the small topology.



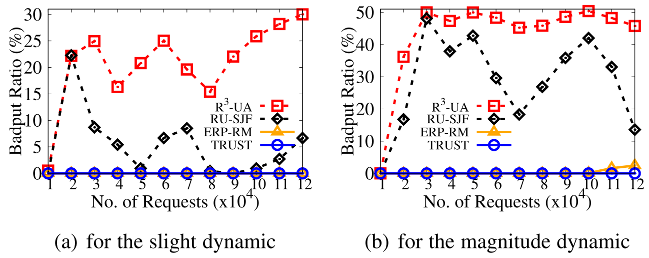
(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 9. Throughput vs. Number of requests in the large topology.



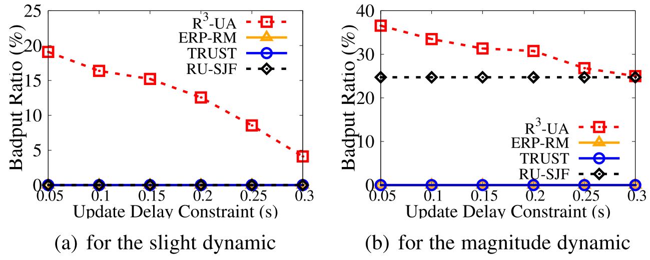
(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 10. Badput ratio vs. Number of requests in the small topology.



(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 11. Badput ratio vs. Number of requests in the large topology.



(a) for the slight dynamic (b) for the magnitude dynamic

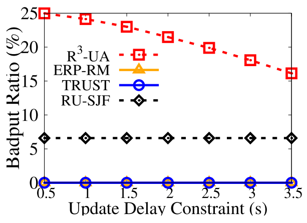
Fig. 12. Badput ratio vs. Update delay constraint in the small topology.

the number of requests in clouds. We can learn from the figures that the throughput results increase as the number of requests grows. By Fig. 6(b), given 12×10^3 requests, TRUST improves throughput by 62.2% and 25.3% compared with R^3-UA and RU-SJF, respectively. In Fig. 7(b), when there are 11×10^4 requests, TRUST improves throughput by 93.2% and 49.3% compared with R^3-UA and RU-SJF, respectively. Since the two methods are only updating requests without buying extra resources, consequently, many requests have to be abandoned due to limited processing capacity on NFs. Note that since ERP-RM and TRUST both tend to buy extra resources when facing dynamics, the throughput results are much better than those of R^3-UA and RU-SJF. However, under some extreme situations, due to the limited resources of physical servers, ERP-RM may not be able to hold all the traffic from users. As comparison, TRUST can balance the load among the servers by updating the requests. As shown in Figs. 7(a)-7(b), when there are 12×10^4 requests under two kinds of traffic dynamics, the throughput results of ERP-RM are a bit lower than those of TRUST.

Figs. 8-9 present the throughputs by varying the update delay constraint. We can conclude that the throughput results of TRUST and ERP-RM are always the highest among the four algorithms. Since both TRUST and ERP-RM adopt elastic resource provisioning, the requests are all served by upgrading the configuration type. As a comparison, the request updating schemes, *i.e.*, RU-SJF and R^3-UA, cannot serve all the requests when facing the traffic dynamics. For instance, in Fig 8(b), the throughput results of TRUST are always 29.27 Gbps. When the update delay constraint is set 0.1s, TRUST improves the throughput by 55.77% and 35.65%, compared with R^3-UA and RU-SJF, respectively. In Fig 9(b), when given 1s, TRUST improves the throughput by 104.47% and 40.13% compared with R^3-UA and RU-SJF, respectively. The throughput results of RU-SJF are always the same when the update delay constraint is changing because RU-SJF does not consider the constraint, while the throughput results of R^3-UA increase when given more time since R^3-UA can update more requests. We should also note that, although ERP-RM achieves the same results as TRUST does, ERP-RM needs to pay more since it does not adopt the request updating, as shown in Fig. 4(b).

Figs. 10-11 present the badput ratio of four algorithms. We can conclude that ERP-RM and TRUST can always achieve the least badput ratio compared with R^3-UA and RU-SJF. We can observe that the function curves of R^3-UA and RU-SJF fluctuate as the number of requests increases. For instance, by Fig. 10(b), the badput ratio of R^3-UA decreases when the number of requests ranges from 3×10^3 to 7×10^3, and increases when the number of requests is between 8×10^3 to 12×10^3, since when the number of requests is ranging from 3×10^3 to 7×10^3, the configuration types of NFs are being upgraded. Consequently, as shown in Fig. 2(b), the cost result of R^3-UA is increasing. In general, TRUST decreases badput by 43% and 28.7% compared with R^3-UA and RU-SJF, respectively, in the large topology.

Figs. 12-13 demonstrate the badput ratio by varying the update delay constraint. We can conclude from the figures that the badput ratio results of TRUST and ERP-RM are always 0 when the update delay constraint is changing. The reason is that TRUST and ERP-RM can upgrade the configuration of NFs when the traffic dynamics occur by elastic resource provisioning. As shown in Fig. 12(b), in the small topology,


(a) for the slight dynamic

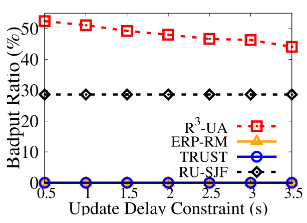
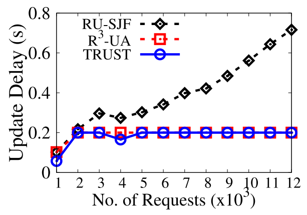

(b) for the magnitude dynamic

Fig. 13. Badput ratio vs. Update delay constraint in the large topology.


(a) for the slight dynamic

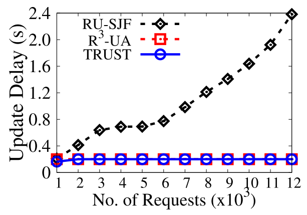
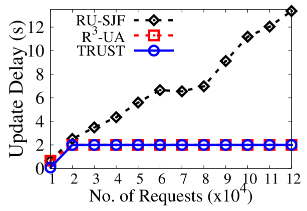

(b) for the magnitude dynamic

Fig. 14. Update delay vs. Number of requests in the small topology.


(a) for the slight dynamic

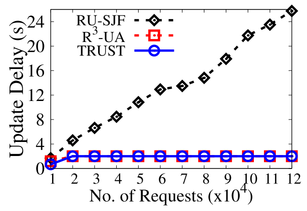
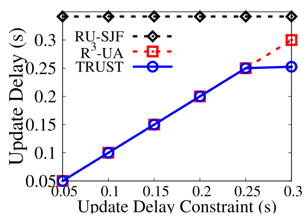

(b) for the magnitude dynamic

Fig. 15. Update delay vs. Number of requests in the large topology.


(a) for the slight dynamic

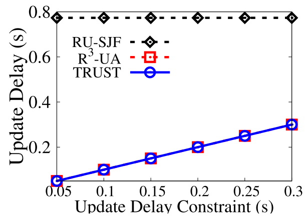
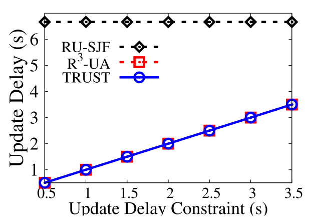

(b) for the magnitude dynamic

Fig. 16. Update delay vs. Update delay constraint in the small topology.


(a) for the slight dynamic

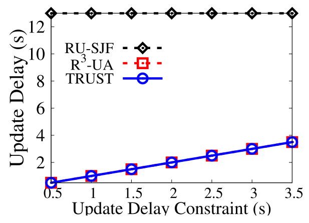
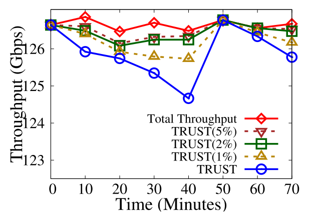

(b) for the magnitude dynamic

Fig. 17. Update delay vs. Update delay constraint in the large topology.


(a) the small topology

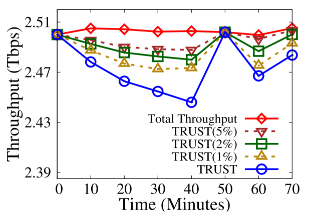
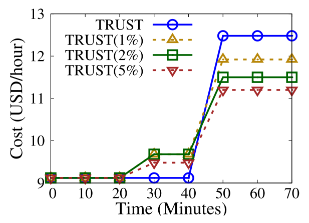

(b) the large topology

Fig. 18. System throughput vs. time.


(a) the small topology

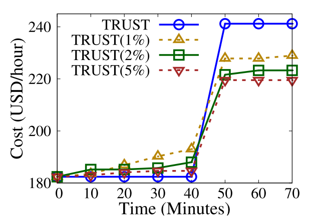

(b) the large topology

Fig. 19. Infrastructure cost vs. Time.

given 0.1s update delay constraint, TRUST can reduce the badput ratio by 33.49% and 24.73% compared with R^3-UA and RU-SJF, respectively. In Fig. 12(b), when the update delay constraint is set 2s, TRUST can reduce 48.02% and 28.64% compared with R^3-UA and RU-SJF, respectively. We can also observe that when given more time, the badput ratio of R^3-UA is decreasing. Although ERP-RM can also achieve 0 badput ratio, it needs to pay more for extra resources than TRUST does.

Figs. 14-15 show the update delay of three algorithms. Since ERP-RM does not involve the updating procedure, we only compare TRUST with R^3-UA and RU-SJF. We learn from Figs. 14-15 that the update delay of RU-SJF will significantly increase while TRUST and R^3-UA limit the update delay within a small range. For instance, by Fig. 15(a), when there are 9×10^4 requests, TRUST can reduce update delay by 78.1% compared with RU-SJF. In Fig. 15(b), when there are 10×10^4 requests, TRUST can reduce update delay by 91.5% compared with RU-SJF.

Figs. 16-17 present the update delay by varying the update delay constraint. We can conclude that, in general, the update delay of TRUST and R^3-UA will increase as the constraint relaxes. Since RU-SJF does not consider the update delay constraint, the update delay of RU-SJF is much higher than that of TRUST. For instance, in Fig. 16(b), when given 0.2s, TRUST can reduce the update delay by 74.14% compared with RU-SJF. We should note that, in particular situations, TRUST can even achieve less update delay than R^3-UA does. In Fig. 16(a), when the update delay constraint is set 0.3s, the actual update delay of TRUST is 0.26s, while that of R^3-UA is 0.3s. The reason is that TRUST combines request updating and elastic resource provisioning. When given appropriate configuration types of NFs, there is no need to update requests as many as R^3-UA does.

In the third group of experiments, we mainly focus on the performance comparison of TRUST and TRUST(x%) discussed in Section III-A. The results are shown in Figs. 18-19. We run TRUST every 50 minutes and TRUST(x%) every 10 minutes, where x is set to 1 and 2. In this group of experiments, we simulate the traffic dynamics according to the following rules. Overall, the total throughput of cloud users tends to be steady [64]. However, for every single request, it may change randomly. We divide requests into the following parts. 1) 25% of requests will keep the original status. 2) 25% of requests will increase at most 1.3 times of current traffic size. 3) 25% of requests' traffic size will decrease by at most 30%. 4) 25% of requests will terminate while nearly the same number of requests will arrive.

The throughput and infrastructure costs are shown in Figs. 18-19. Here total throughput means the total traffic generated by cloud users. In simulations, some traffic may be denied due to traffic dynamics and overload NFs. Thus, it is necessary to test the throughput performance when running different schemes, *i.e.*, TRUST and TRUST(x%). From Fig. 18, although throughput performance will get worse along with time due to traffic dynamics, TRUST ($x\%$) can further promote the throughput performance. For instance, in Fig. 18(a), at the 30th minutes, TRUST(1%), TRUST(2%) and TRUST(5%) improve throughput by 0.72%, 1.14% and 1.38%, respectively, compared with TRUST. The reason is that if we execute TRUST($x\%$) between the interval of TRUST, throughput can be improved by updating a part of requests and upgrading NFs. Correspondingly, in Fig. 19(a), the costs of TRUST ($x\%$) also increase compared with TRUST. For instance, at the 30th minutes, both TRUST(1%) and TRUST(2%) increase costs by 5.79% compared with TRUST, and TRUST(5%) increase cost by 3.95% compared with TRUST. However, after we execute TRUST for a global update at the 50th minutes, the infrastructure costs of all TRUST($x\%$) are fewer than that of TRUST. Since TRUST($x\%$) updates requests frequently and there are fewer overloaded NFs, there is no need to upgrade the NFs as many as TRUST does. The reason why TRUST(1%) costs more than TRUST(2%) does is that TRUST(2%) involves more updated requests and NFs. That is to say, with TRUST(2%), NFs tend to be more load-balancing compared with TRUST(1%). Therefore, when we execute Algorithm 1 for a global update at the 50th minutes, TRUST(2%) can acquire a lower infrastructure cost compared with TRUST(1%).In conclusion, we can update the cloud with less cost compared with TRUST.

From these simulation results, we can draw some conclusions. Firstly, compared with ERP-RM, TRUST can significantly reduce the cost by 35.5%/31.3% on average in the small/large topology by Figs. 2-3. At the same time, TRUST only takes a few more seconds of update delay, and it can still satisfy the QoS demand of users. Figs. 4-5 show that the cost of TRUST can reduce as the update delay constraint relaxes. Secondly, compared with R^3-UA, TRUST has a much better performance on throughput and significantly reduces the badput ratio. From Figs. 6-7, we know that TRUST improves the throughput by 44.9%/88.9% in the small/large topology on average compared with R^3-UA. When the update delay constraint is changing, the throughput results of TRUST would still be the same, as presented in Figs. 8-9. Then, Figs. 10-11 show that TRUST can reduce the badput ratio by 29.1%/43% in the small/large topology compared with R^3-UA. Meanwhile, TRUST only increases the cost by about 13.79%/8.8% in the small/large topology compared with R^3-UA. And we can see that when the update delay constraint changes, the badput ratio of TRUST is always zero, as presented in Figs. 12-13. Finally, compared with RU-SJF, TRUST can greatly reduce the update delay. Figs. 14-15 show that in general, TRUST can reduce 81.8%/86% update delay compared with RU-SJF in the small/large topology. Also, TRUST can improve the throughput by 15%-44.8% and reduce the badput ratio by 13.9%-28.7% with cost increased only by 8.8%-13.79% compared with RU-SJF. As the update delay constraint relaxes, the actual update delay of TRUST would also increase, as shown in Figs. 16-17. However, there are still particular situations where TRUST achieves less update delay than the constraint. Last but not least, compared with TRUST, TRUST(1%) and TRUST(2%) can further improve the throughput performance and reduce the cost.

C. Testbed Evaluation

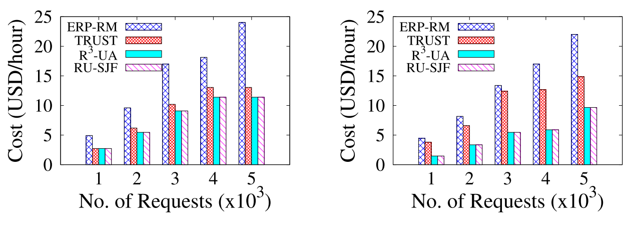
This section implements TRUST on a small-scale testbed to evaluate the performance of these algorithms.

1) Testbed Settings: To better evaluate our proposed algorithm, we need a universal platform to realize our small-scale testbed, which should be similar to the real-world cloud environment. To this end, we choose OpenStack, the most advanced and widely used cloud infrastructure software, to implement the system. In general, we adopt the latest version of OpenStack called Yoga [65] to estimate our algorithm. Using OpenStack's VM service Nova, we can customize different *flavors*, *i.e.*, configuration types, according to Google Cloud Platform [29], which is shown in Table III. We adopt Open vSwitch (OvS) of version 2.17.3 as SDN switches, which is a default software switch of OpenStack platform. The bridge in OvS used in our experiments is br-int. We use the OpenFlow protocol of version 1.5 to control the flow entries to direct flows. The controller is based on Neutron.

In our testbed, we generate 20 instances as NFs, all with Ubuntu 18.04 OS. The NFs are all initialized as type 1 in Table III. We set 3000 requests and 0.1s update delay constraint by default. We first run SNAT (Static Network Address Translation) on 20 NFs by rewriting the iptables rules to translate the private address into a public address for online requirements. Then, we set flow data as the same as the simulations. We also generate an instance for sending requests and an instance as the controller. First, we use iPerf3 to generate request data. Then, we simulate the dynamics according to the rule of magnitude dynamics. After the 20 NFs experiencing load-unbalancing, we acquire the joint updating solutions by running the algorithm. Finally, the controller sends new rules to the corresponding NF to reroute the request, and new configurations are also updated. Then we clarify how we execute some important procedures.

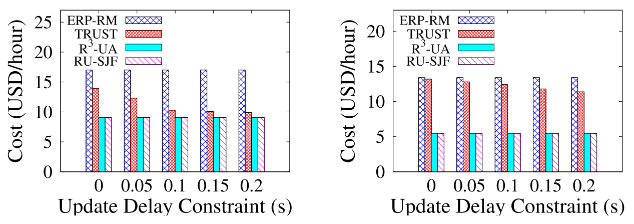
How we apply scaling to NFs: OpenStack provides a computing service called Nova to help create and manage NFs. It allows us to customize flavor, *i.e.*, different configurations for NFs. For instance, Nova allows users to specify RAM, vCPU, disk volume and other parameters in a flavor. When creating an NF, a flavor must be specified. Due to traffic dynamics, the flavor of an NF may be inappropriate along with time. Thus, Nova also provides interface to resize an NF, *i.e.*, the command *OpenStack server resize NF-NAME Flavor-NAME* [66]. After acquiring the algorithm results, we can execute the corresponding command to resize NFs from current configuration to the new configuration. With the advancement of work [67], the resizing duration can only take from sub-one millisecond to the order of four milliseconds, which is negligible compared with requests updating.

How we redirect flows: We design the controller based on *Neutron*, the network managing service provided by OpenStack, responsible for sending/modifying flow rules. The specific flow rules setting is done by Neutron-openvswitch-agent. Next, we clarify how we redirect flows. First, we use iPerf3 to create and send flows. Each flow is corresponding to one flow entry stored in br-int. Flow entry will check the matching field (*e.g.*, source ip and source port) to determine the corresponding action, *i.e.*, choosing which NF to send the flow. For instance, flow 1 is sent from source port 12345. Then, in the matching field, the corresponding action determines that it will be sent to NF_2. After updating, the algorithm decides that flow 1 needs to be redirected to NF_3. Correspondingly, we use RESTful API to send results to Neutron server by HTTP. Then, Neutron server would use message queue (RabbitMQ) to modify the corresponding rules by sending flow-mod messages to Neutron-openvswitch-agent, which are mainly ovs-ofctl



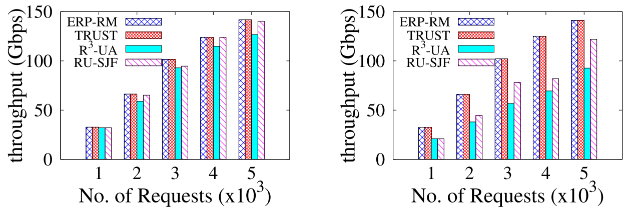
(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 20. Cost vs. Number of requests.



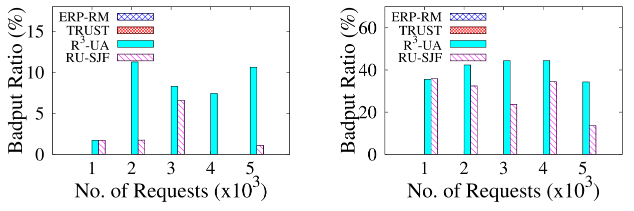
(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 21. Cost vs. Update delay constraint.



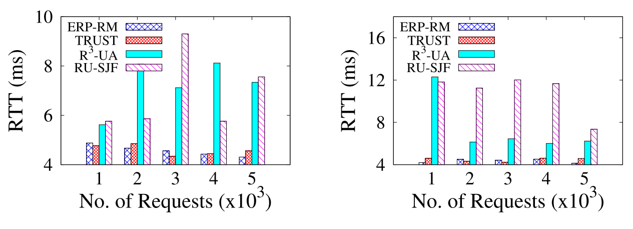
(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 22. Throughput vs. Number of requests.



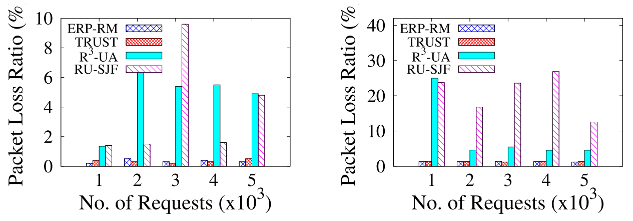
(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 23. Badput ratio vs. Number of requests.



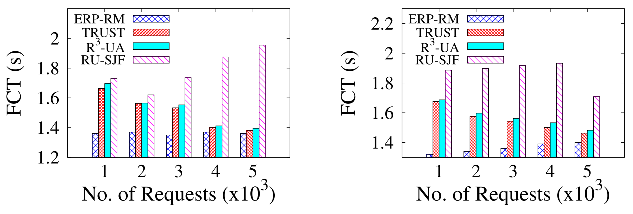
(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 24. RTT vs. Number of requests.



(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 25. Packet loss ratio vs. Number of requests.



(a) for the slight dynamic (b) for the magnitude dynamic

Fig. 26. FCT vs. Number of requests.

commands [68]. Finally, it will be redirected to the correct NF.

How we set up the switches, the controller and time threshold: We adopt Open vSwitch (OvS) of version 2.17.3 as SDN switches, which is a default software switch of OpenStack platform. The controller is based on Neutron. When the algorithm for TRUST acquires an update solution, the controller will generate corresponding messages and commands. The time threshold is an input parameter of the algorithm. The setting of the time threshold accords to the network status, 0.1s by default in our experiments. Then we can use current network information (*e.g.*, NFs configuration, NFs load and traffic size) to execute an algorithm for joint elasticity scaling and request updating solutions.

2) Performance Comparison: We run six sets of testbed experiments to compare the performance of ERP-RM, TRUST, R^3-UA and RU-SJF, and the results are shown in Figs. 20-26.

The first set of experiments evaluates the infrastructure cost by varying the number of requests and the update delay constraint. We record each NF's configuration type and calculate the total infrastructure cost and the results are shown in Figs. 20 and 21. We observe that in Fig. 20(a), given 3×10^3 requests, TRUST can reduce the cost by 41.17% compared with ERP-RM. Since TRUST takes the advantage of request updating, the purchased resources of TRUST are much fewer than that of ERP-RM. In Fig. 20(b), given 4×10^3 requests, TRUST reduces the cost by 25.41% compared with ERP-RM. In Fig. 21, we show the infrastructure cost by varying the update delay constraint. We can infer that in general, the cost results of TRUST decrease as the update delay constraint relaxes. For example, in both Figs. 21(a) and 21(b), when the update delay is 0, the cost result of TRUST is almost the same as that of ERP-RM since TRUST is unable to transfer any requests. In Fig. 21(a), given 0.2s update delay, TRUST can reduce 41.71% infrastructure cost compared with ERP-RM and only increase cost by 9.14% compared with both R^3-UA and RU-SJF. In Fig. 21(b), however, TRUST cannot significantly reduce cost when given enough update delay. This is because when facing the magnitude dynamics, we always need to buy extra resources.

The second set of experiments compares the throughput performance by varying the number of requests, and the results are shown in Fig. 22. In Fig. 22(a), we can observe that the throughput performance of TRUST is similar to that of R^3-UA and RU-SJF. For instance, when there are 3×10^3 requests, TRUST only improves the throughput by 9.05% and 7.06% compared with R^3-UA and RU-SJF, respectively. As comparison, when facing the magnitude, in Fig. 22(b), TRUST outperforms R^3-UA and RU-SJF greatly. Given 4×10^3 requests, TRUST improves throughput by 80.03% and 52.59% compared with R^3-UA and RU-SJF, respectively. Since the two request updating schemes do not upgrade the configuration, NFs have to abandon many requests beyond their capacity, resulting in a bad throughput.

The third set of experiments demonstrates the badput ratio performance by varying the number of requests and the results are shown in Fig. 23. We can learn that since both ERP-RM and TRUST adopt the elastic resource provisioning, NFs can upgrade the configuration type during traffic dynamics. Thus, the badput ratio results are always 0. As a comparison, the badput ratio results of R^3-UA and RU-SJF are changing when the number of requests grows. For instance, in Fig. 23(a), given 3×10^3 requests, TRUST reduces the badput ratio by 8.30% and 6.59% compared with R^3-UA and RU-SJF, respectively. And in Fig. 23(b), given 4×10^3 requests, TRUST reduces the badput ratio by 44.45% and 34.47% compared with R^3-UA and RU-SJF, respectively.

The fourth set of experiments compares the average RTT of NFs by varying the number of requests and the results are shown in Fig. 24. We can see the average RTT results of ERP-RM and TRUST are always the lowest since the two schemes will buy extra resources when the current capacity of NFs cannot hold requests. As a comparison, the request updating schemes, *i.e.*, R^3-UA and RU-SJF, cannot solve the insufficient capacity problem. Consequently, there may exist many NFs overloaded. In Fig. 24(a), when given 5×10^3 requests, TRUST can reduce the average RTT by 37.87% and 39.68% compared with R^3-UA and RU-SJF, respectively. In Fig. 24(b), when there are 1×10^3 requests, TRUST reduces the average RTT by 62.6% and 61.08% compared with R^3-UA and RU-SJF, respectively.

The fifth set of experiments compares the average packet loss ratio by varying the number of requests and the results are shown in Fig. 25. Generally, the average packet loss ratio results of ERP-RM and TRUST are always the lowest among the four schemes, since both two methods tend to buy enough resources to handle requests. In Fig. 25(a), when the number of requests is 3×10^3, TRUST can reduce 91.7% and 90.3% packet loss ratio compared with R^3-UA and RU-SJF, respectively. And in Fig. 25(b), when the number of requests are 1×10^3, TRUST can reduce 94.4% and 94.11% packet loss ratio compared with R^3-UA and RU-SJF, respectively.

The sixth set of experiments compares the average flow completion time (FCT) of requests by varying the number of requests and the results are shown in Fig. 26. The FCT of a single request mainly depends on its traffic size and link bandwidth. If the request is transferred, the time for updating rules is also included. The FCT results of ERP-RM are always the lowest because ERP-RM does not involve updating the requests. Overall, the FCT results of TRUST and R^3-UA decrease as the number of requests grows, since the update delay constraint is fixed, *i.e.*, the number of requests is also constrained. Consequently, the average FCT will decrease as the number of total requests increases. In Fig. 26(a), when given 3×10^3 requests in the cloud, TRUST can reduce the average FCT by 12.56% compared with RU-SJF and only increase the average FCT by about 10.76% compared with ERP-RM. And in Fig. 26(b), TRUST can reduce the average FCT by 22.34% compared with RU-SJF. We can also see from the figure that, when the number of requests is 5×10^3, the FCT of RU-SJF significantly decreases, since at this time, the configuration of NFs have been upgraded, which can be inferred from Fig. 20(b). The reason why TRUST and R^3-UA can achieve a much lower FCT than that of RU-SJF, is that TRUST and R^3-UA can limit the number of updating requests.

From the above experimental results, we can draw some conclusions. Firstly, TRUST reduces the cost by 41.71% and only increases the average FCT by 10.76% compared with ERP-RM. Secondly, TRUST improves the throughput performance by 80.03% and 52.59% compared with R^3-UA and RU-SJF, respectively. And TRUST reduces the badput ratio by 44.45% and 34.47% compared with R^3-UA and RU-SJF, respectively. Thirdly, compared with R^3-UA and RU-SJF, TRUST can acquire a much better RTT and packet loss ratio result. TRUST reduces the average RTT by 37.87% and 39.68% compared with R^3-UA and RU-SJF, respectively. And TRUST reduces the average packet loss ratio by 91.7% and 90.3% compared with R^3-UA and RU-SJF, respectively. Finally, TRUST can reduce the average FCT by 22.34% compared with RU-SJF. These experimental results show the high efficiency and cost-saving of TRUST.

Another approach to deal with traffic dynamics, *i.e.*, request updating, usually focuses on the load-balancing [10], [11], [75], [76] or makespan [13], [77], [78]. For instance, the work [75] proposes a distributed and adaptive algorithm for load balancing in data center networks when facing random especially dynamic traffic patterns. And the work [77] proposes an algorithm with an arbitrarily-good approximation factor to schedule requests on a multiprocessor achieving the optimal makespan. However, all these works cannot avoid the delay caused by the update, which may severely damage the users' QoS. Furthermore, frequent update may affect the consistency of requests. Thus, we believe request updating and elastic resource provisioning can be complementary to each other, to help service providers save more cost while guaranteeing users' QoS.

V. Related Works

As the virtualization technique develops, both academia and industry have been exploring elasticity in cloud computing from all aspects. In general, current elastic resource provisioning has two methods: reactive mode [36], [69], [70], [71] and proactive mode [30], [72], [73], [74]. Specifically, reactive mode means there exist several certain thresholds or rules to trigger the elastic resource provisioning. For instance, when reaching a specific amount of workload or resource utilization, the traffic dynamics will be detected and elastic actions will be executed. Overall, the thresholds can be divided into two kinds: static and dynamic. Static thresholds, like [36] and [69] will monitor one or more metrics, when the threshold is reached, the configuration will be upgraded. The other scheme adopts the dynamic threshold [70], [71], which is more agile and efficient. Specifically, the thresholds change dynamically according to the state of VMs or NFs. The work [70] adopts the dynamic CPU utilization as the thresholds. Proactive mode, as a comparison, adopts forecasting techniques to anticipate the potential future resource utilization and execute elastic actions based on the anticipation. The work [72] and [73] both adopt the time series analysis technique while [30], [74] use reinforcement learning to make decisions. However, we should note that no matter what mode we adopt, we always need to pay for the extra resources we used. Thus, combining the request updating technique is an excellent complementary work to save cost.

Another approach to deal with traffic dynamics, *i.e.*, request updating, usually focuses on the load-balancing [10], [11],

[75], [76] or makespan [13], [77], [78]. For instance, the work [75] proposes a distributed and adaptive algorithm for load balancing in data center networks when facing random especially dynamic traffic patterns. And the work [77] proposes an algorithm with an arbitrarily-good approximation factor to schedule requests on a multiprocessor achieving the optimal makespan. However, all these works cannot avoid the delay caused by the update, which may severely damage the users' QoS. Furthermore, frequent update may affect the consistency of requests. Thus, we believe request updating and elastic resource provisioning can be complementary to each other, to help service providers save more cost while guaranteeing users' QoS.

VI. Conclusion

In this paper, we focus on the problem of real-time request updating with elastic resource provisioning in clouds. To solve the problem, we design an efficient algorithm with bounded approximation factors based on progressive-rounding. Extensive simulation and testbed experiments results show the high efficiency of our proposed algorithm.

References

[1] J. Wang, G. Zhao, H. Xu, Y. Zhao, X. Yang, and H. Huang, "TRUST: Real-time request updating with elastic resource provisioning in clouds," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2022, pp. 620–629.

[2] M. M. Sadeeq, N. M. Abdulkareem, S. R. M. Zeebaree, D. M. Ahmed, A. S. Sami, and R. R. Zebari, "IoT and cloud computing issues, challenges and opportunities: A review," *Qubahan Academic J.*, vol. 1, no. 2, pp. 1–7, Mar. 2021.

[3] I. M. Ibrahim, "Task scheduling algorithms in cloud computing: A review," *Turkish J. Comput. Math. Educ. (TURCOMAT)*, vol. 12, no. 4, pp. 1041–1053, Apr. 2021.

[4] M. Armbrust et al., "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[5] B. Andrei, "Threat modeling of cloud systems with ontological security pattern catalog," *Int. J. Open Inf. Technol.*, vol. 9, no. 5, pp. 36–41, 2021.

[6] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," *VLDB J.*, vol. 30, no. 1, pp. 25–43, Jan. 2021.

[7] Y. Zhang, G. Lin, H. Gu, F. Zhuang, and G. Wei, "Multi-copy dynamic cloud data auditing model based on IMB tree," *Enterprise Inf. Syst.*, vol. 15, no. 2, pp. 248–269, Feb. 2021.

[8] Y. Zhou, L. Ruan, L. Xiao, and R. Liu, "A method for load balancing based on software defined network," *Adv. Sci. Technol. Lett.*, vol. 45, pp. 43–48, Feb. 2014.

[9] A. Beloglazov and R. Buyya, "Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 7, pp. 1366–1379, Jul. 2013.

[10] S. Bharti and K. K. Pattanaik, "Dynamic distributed flow scheduling with load balancing for data center networks," *Proc. Comput. Sci.*, vol. 19, pp. 124–130, Jan. 2013.

[11] Z. Guo et al., "AggreFlow: Achieving power efficiency, load balancing, and quality of service in data center networks," *IEEE/ACM Trans. Netw.*, vol. 29, no. 1, pp. 17–33, Feb. 2021.

[12] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading tasks with dependency and service caching in mobile edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2777–2792, Nov. 2021.

[13] P. Wang, Y. Lei, P. R. Agbedanu, and Z. Zhang, "Makespan-driven workflow scheduling in clouds using immune-based PSO algorithm," *IEEE Access*, vol. 8, pp. 29281–29290, 2020.

[14] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. IEEE ICNN*, vol. 4, Nov./Dec. 1995, pp. 1942–1948.

[15] P. Berde et al., "ONOS: Towards an open, distributed SDN OS," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.*, 2014, pp. 1–6.

[16] F. Yamei, L. Qing, and H. Qi, "Research and comparative analysis of performance test on SDN controller," in *Proc. 1st IEEE Int. Conf. Comput. Commun. Internet (ICCCI)*, Oct. 2016, pp. 207–210.

[17] P. Wang, H. Xu, L. Huang, C. Qian, S. Wang, and Y. Sun, "Minimizing controller response time through flow redirecting in SDNs," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 562–575, Feb. 2018.

[18] M. Dolati, A. Khonsari, and M. Ghaderi, "Minimizing update makespan in SDNs without TCAM overhead," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 2, pp. 1598–1613, Jun. 2022.

[19] G. Li, Y. R. Yang, F. Le, Y. Lim, and J. Wang, "Update algebra: Toward continuous, non-blocking composition of network updates in SDN," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 1081–1089.

[20] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 323–334, Aug. 2012, doi: 10.1145/2377677.2377748.

[21] C.-Y. Hong et al., "Achieving high utilization with software-driven WAN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 15–26, Aug. 2013, doi: 10.1145/2534169.2486012.

[22] H. H. Gharakheili, M. Lyu, Y. Wang, H. Kumar, and V. Sivaraman, "ITeleScope: Softwarized network middle-box for real-time video telemetry and classification," *IEEE Trans. Netw. Service Manage.*, vol. 16, no. 3, pp. 1071–1085, Sep. 2019.

[23] W. J. A. Silva, "Avoiding inconsistency in OpenFlow stateful applications caused by multiple flow requests," in *Proc. Int. Conf. Comput., Netw. Commun. (ICNC)*, Mar. 2018, pp. 548–553.

[24] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of resource provisioning cost in cloud computing," *IEEE Trans. Services Comput.*, vol. 5, no. 2, pp. 164–177, Jun. 2012.

[25] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Proc. 31st Int. Conf. Distrib. Comput. Syst.*, Jun. 2011, pp. 559–570.

[26] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic provisioning of virtual machines for container deployment," in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. Companion*, Apr. 2017, pp. 5–10.

[27] A. da Silva Dias, L. H. V. Nakamura, J. C. Estrella, R. H. C. Santana, and M. J. Santana, "Providing IaaS resources automatically through prediction and monitoring approaches," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2014, pp. 1–7.

[28] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 430–447, Mar. 2018.

[29] *Google Cloud Platform*. Accessed: Jun. 1, 2023. [Online]. Available: https://cloud.google.com/compute/vm-instance-pricing

[30] A. Ashraf, B. Byholm, and I. Porres, "CRAMP: Cost-efficient resource allocation for multiple web applications with proactive scaling," in *Proc. 4th IEEE Int. Conf. Cloud Comput. Technol. Sci.*, Dec. 2012, pp. 581–586.

[31] P. Marshall, K. Keahey, and T. Freeman, "Elastic site: Using clouds to elastically extend site resources," in *Proc. 10th IEEE/ACM Int. Conf. Cluster, Cloud Grid Comput.*, May 2010, pp. 43–52.

[32] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic resource scaling for multi-tenant cloud systems," in *Proc. 2nd ACM Symp. Cloud Comput.*, Oct. 2011, pp. 1–14.

[33] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. 15th {USENIX} Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 299–312.

[34] M. Karzand, D. J. Leith, J. Cloud, and M. Médard, "Design of FEC for low delay in 5G," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 8, pp. 1783–1793, Aug. 2017.

[35] H. Xu, Z. Yu, X. Li, L. Huang, C. Qian, and T. Jung, "Joint route selection and update scheduling for low-latency update in SDNs," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3073–3087, Oct. 2017.

[36] S. M.-K. Gueye, N. D. Palma, É. Rutten, A. Tchana, and N. Berthier, "Coordinating self-sizing and self-repair managers for multi-tier systems," *Future Gener. Comput. Syst.*, vol. 35, pp. 14–26, Jun. 2014.

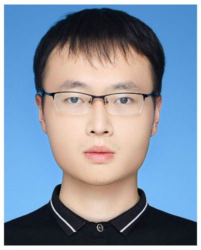
[37] H. Tu, G. Zhao, H. Xu, Y. Zhao, and Y. Zhai, "Robustness-aware real-time SFC routing update in multi-tenant clouds," in *Proc. IEEE/ACM 29th Int. Symp. Quality Service (IWQOS)*, Jun. 2021, pp. 1–6.

[38] M. Nosrati, R. Karimi, and M. Hariri, "Task scheduling algorithms introduction," *World Appl. Program.*, vol. 2, no. 6, pp. 394–398, 2017.

[39] W. Yu, L. Musavian, and Q. Ni, "Link-layer capacity of NOMA under statistical delay QoS guarantees," *IEEE Trans. Commun.*, vol. 66, no. 10, pp. 4907–4922, Oct. 2018.

[40] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot, “Traffic matrix estimation: Existing techniques and new directions,” in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Aug. 2002, pp. 161–174, doi: 10.1145/633025.633041.
[41] A. Gunnar, M. Johansson, and T. Telkamp, “Traffic matrix estimation on a large IP backbone: A comparison on real data,” in *Proc. 4th ACM SIGCOMM Conf. Internet Meas.*, Oct. 2004, pp. 149–160, doi: 10.1145/1028788.1028807.
[42] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, “OpenTM: Traffic matrix estimator for openflow networks,” in *Passive and Active Measurement*, A. Krishnamurthy and B. Plattner, Eds. Berlin, Germany: Springer, 2010, pp. 201–210.
[43] X. Yu, H. Xu, D. Yao, H. Wang, and L. Huang, “CountMax: A lightweight and cooperative sketch measurement for software-defined networks,” *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2774–2786, Dec. 2018.
[44] H. Xu, Z. Yu, X. Li, C. Qian, L. Huang, and T. Jung, “Real-time update with joint optimization of route selection and update scheduling for SDNs,” in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2016, pp. 1–10.
[45] H. Jahanjou, R. Rajaraman, and D. Stalfa, “Scheduling flows on a switch to optimize response times,” in *Proc. 32nd ACM Symp. Parallelism Algorithms Architectures*, Jul. 2020, pp. 305–315, doi: 10.1145/3350755.3400218.
[46] G. Patti, L. L. Bello, and L. Leonardi, “Deadline-aware online scheduling of TSN flows for automotive applications,” *IEEE Trans. Ind. Informat.*, vol. 19, no. 4, pp. 5774–5784, Apr. 2023.
[47] Q. Zhou, P. Li, K. Wang, D. Zeng, S. Guo, and M. Guo, “Swallow: Joint online scheduling and coflow compression in datacenter networks,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2018, pp. 505–514.
[48] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, “ZUpdate: Updating data center networks with zero loss,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 411–422, Aug. 2013, doi: 10.1145/2534169.2486005.
[49] C. Huang, J. Zhang, and T. Huang, “Updating data-center network with ultra-low latency data plane,” *IEEE Access*, vol. 8, pp. 2134–2144, 2020.
[50] J. Fang, G. Zhao, H. Xu, C. Wu, and Z. Yu, “GRID: Gradient routing with in-network aggregation for distributed training,” *IEEE/ACM Trans. Netw.*, early access, Feb. 22, 2023, doi: 10.1109/TNET.2023.3244794.
[51] X. Wen et al., “RuleTris: Minimizing rule update latency for TCAM-based SDN switches,” in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2016, pp. 179–188.
[52] S. Martello and P. Toth, “Bin-packing problem,” in *Knapsack Problems: Algorithms Computer Implementations*. Hoboken, NJ, USA: Wiley, 1990, pp. 221–245.
[53] S. Even, A. Itai, and A. Shamir, “On the complexity of time table and multi-commodity flow problems,” in *Proc. 16th Annu. Symp. Found. Comput. Sci. (SFCS)*, Oct. 1975, pp. 184–193.
[54] B. Heller et al., “Elastictree: Saving energy in data center networks,” in *Proc. USENIX NSDI*, vol. 10, 2010, pp. 249–264.
[55] Y. Xu et al., “Dynamic switch migration in distributed software-defined networks to achieve controller load balance,” *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 515–529, Mar. 2019.
[56] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, “Autonomic vertical elasticity of Docker containers with ELASTICDOCKER,” in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2017, pp. 472–479.
[57] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, “Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications,” in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2018, pp. 82–89.
[58] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, “Towards energy proportionality for large-scale latency-critical workloads,” in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 301–312.
[59] P. Raghavan and C. D. Tompson, “Randomized rounding: A technique for provably good algorithms and algorithmic proofs,” *Combinatorica*, vol. 7, no. 4, pp. 365–374, Dec. 1987.
[60] *Pulp*. Accessed: Jun. 1, 2023. [Online]. Available: https://pypi.org/project/PuLP/
[61] V. Apte, “‘What did I learn in performance analysis last year?’: Teaching queuing theory for long-term retention,” in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, Mar. 2019, pp. 71–77.
[62] *Google Cluster Data*. Accessed: Jun. 1, 2023. [Online]. Available: https://www.github.com/google/cluster-data
[63] A. Ali-Eldin, O. Seleznjev, S. Sjöstedt-de Luna, J. Tordsson, and E. Elmroth, “Measuring cloud workload burstiness,” in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput.*, Dec. 2014, pp. 566–572.
[64] W. Shi and B. Hong, “Resource allocation with a budget constraint for computing independent tasks in the cloud,” in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, Dec. 2010, pp. 327–334.
[65] *Openstack Yoga*. Accessed: Jun. 1, 2023. [Online]. Available: https://docs.openstack.org/Yoga/
[66] *Nova*. Accessed: Jun. 1, 2023. [Online]. Available: https://docs.openstack.org/newton/user-guide/cli-change-the-size-of-your-server.html
[67] S. Heuchert, B. P. Rimal, M. Reisslein, and Y. Wang, “Design of a small-scale and failure-resistant IaaS cloud using OpenStack,” *Appl. Comput. Inform.*, Sep. 2021, doi: 10.1108/ACI-04-2021-0094.
[68] *Neutron*. Accessed: Jun. 1, 2023. [Online]. Available: https://docs.openstack.org/api-ref/network/index.html
[69] P. D. Kaur and I. Chana, “A resource elasticity framework for QoS-aware execution of cloud applications,” *Future Gener. Comput. Syst.*, vol. 37, pp. 14–25, Jul. 2014.
[70] L. M. Vaquero, D. Morán, F. Galán, and J. M. Alcaraz-Calero, “Towards runtime reconfiguration of application control policies in the cloud,” *J. Netw. Syst. Manage.*, vol. 20, no. 4, pp. 489–512, Dec. 2012.
[71] A. Beloglazov and R. Buyya, “Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers,” *MGC@ Middleware*, vol. 4, Dec. 2010, Art. no. 1890799.
[72] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, “Agile dynamic provisioning of multi-tier internet applications,” *ACM Trans. Auto. Adapt. Syst.*, vol. 3, no. 1, pp. 1–39, Mar. 2008.
[73] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, “A hybrid reinforcement learning approach to autonomic resource allocation,” in *Proc. IEEE Int. Conf. Autonomic Comput.*, Jun. 2006, pp. 65–73.
[74] E. Kalyvianaki, T. Charalambous, and S. Hand, “Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters,” in *Proc. 6th Int. Conf. Autonomic Comput.*, Jun. 2009, pp. 117–126.
[75] C.-M. Cheung and K.-C. Leung, “DFFR: A flow-based approach for distributed load balancing in data center networks,” *Comput. Commun.*, vol. 116, pp. 1–8, Jan. 2018.
[76] S. Bharti and K. K. Pattanaik, “Dynamic distributed flow scheduling for effective link utilization in data center networks,” *J. High Speed Netw.*, vol. 20, no. 1, pp. 1–10, 2014.
[77] D. P. Bunde, “Power-aware scheduling for makespan and flow,” *J. Scheduling*, vol. 12, no. 5, pp. 489–500, Oct. 2009.
[78] R. Aggoune, “Minimizing the makespan for the flow shop scheduling problem with availability constraints,” *Eur. J. Oper. Res.*, vol. 153, no. 3, pp. 534–543, Mar. 2004.

Gongming Zhao (Member, IEEE) received the Ph.D. degree in computer software and theory from the University of Science and Technology of China in 2020. He is currently an Associate Professor with the University of Science and Technology of China. His current research interests include software defined networks and cloud computing.



Jingzhou Wang (Graduate Student Member, IEEE) is currently pursuing the Ph.D. degree in computer science with the University of Science and Technology of China. His main research interests include software defined networks and cloud computing.

Hongli Xu (Member, IEEE) received the B.S. degree in computer science and the Ph.D. degree in computer software and theory from the University of Science and Technology of China (USTC), China, in 2002 and 2007, respectively. He is currently a Professor with the School of Computer Science and Technology, USTC. He has published more than 100 articles in famous journals and conferences, including IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, International Conference on Computer Communications (INFOCOM), and International Conference on Network Protocols (ICNP). He has held more than 30 patents. His research interests include software defined networks, edge computing, and the Internet of Things. He was awarded the Outstanding Youth Science Foundation of NSFC in 2018. He has won the best paper award and the best paper candidate in several famous conferences.

Yangming Zhao (Member, IEEE) received the B.S. degree in communication engineering and the Ph.D. degree in communication and information system from the University of Electronic Science and Technology of China, in 2008 and 2015, respectively. He is a Research Professor with the School of Computer Science and Technology, University of Science and Technology of China. Before that, he was a Research Scientist with University at Buffalo. His research interests include network optimization, quantum networks, edge computing, and machine learning.

Xuwei Yang received the Ph.D. degree in computer software and theory from the University of Science and Technology of China in 2021. He is currently a Software Engineer with Huawei Cloud Computing Technology Company Ltd. His current research interests include software defined networks, cloud computing, and network function virtualization.

He Huang (Member, IEEE) received the Ph.D. degree from the School of Computer Science and Technology, University of Science and Technology of China (USTC), China, in 2011. From 2019 to 2020, he was a Visiting Research Scholar with Florida University, Gainesville, FL, USA. He is currently a Professor with the School of Computer Science and Technology, Soochow University, China. He has authored more than 100 papers in related international conference proceedings and journals. His current research interests include traffic measurement, computer networks, and algorithmic game theory. He is a member of the Association for Computing Machinery (ACM). He has served as the Technical Program Committee Member for several conferences, including IEEE INFOCOM, IEEE MASS, IEEE ICC, and IEEE Globecom. He received the Best Paper Awards from Bigcom 2016, IEEE MSN 2018, and Bigcom 2018.