Virtual Service Embedding with Time-Varying Load and Provable Guarantees

Virtual Service Embedding with Time-Varying Load and Provable Guarantees

Gil Einziger, *Member, IEEE,* Gabriel Scalosub, *Senior Member, IEEE,* Carla Fabiana Chiasserini, *Fellow, IEEE,* Francesco Malandrino, *Senior Member, IEEE*

***Abstract*—Deploying services efficiently while satisfying their quality requirements is a major challenge in network slicing. Effective solutions place instances of the services' virtual network functions (VNFs) at different locations of the cellular infrastructure and manage such instances by scaling them as needed. In this work, we address the above problem and the very relevant aspect of sub-slice reuse among different services. Further, unlike prior art, we account for the services' finite lifetime and time-varying traffic load. We identify two major sources of inefficiency in service management: (i) the overspending of computing resources due to traffic of multiple services with different latency requirements being processed by the same virtual machine (VM), and (ii) the poor packing of traffic processing requests in the same VM, leading to opening more VMs than necessary. To cope with the above issues, we devise an algorithm, called REShare, that can dynamically adapt to the system's operational conditions and find an optimal trade-off between the aforementioned opposite requirements. We prove that REShare has low algorithmic complexity and is asymptotic 2-competitive under a non-decreasing load. Numerical results, leveraging real-world scenarios, show that our solution outperforms alternatives, swiftly adapting to time-varying conditions and reducing service cost by over 25%.**

I. Introduction

Network slicing leads to a revolutionary transformation of mobile services, with technologies like software-defined networking (SDN) and network function virtualization (NFV) enabling flexible, fully virtualized environments. In this context, the automated management of the services that the network supports and of the underlying resources they consume is a major challenge. As requests for service deployment can arrive and leave at a very fast pace, it is important that deployment decisions are swift and able to *dynamically* adapt to the evolution of the network load. Additionally, each such deployment has to fulfill the services' target key performance indicators (KPIs) and efficiently allocate the very diverse, geographically distributed, and differently owned resources.

Such requirements imply that when a service request from a third-party *vertical* (e.g., automotive industry or a content provider) reaches the system, the following steps should be automatically performed: (i) to identify the network segment (e.g., edge, aggregation, cloud) where to deploy the service, based on the service target latency and the infrastructure cost, (ii) to check whether part (or all) of the virtual network functions (VNFs) composing the service can re-use already deployed VNFs (i.e., (sub-)slices), (iii) if so, to scale the amount of resources allocated for the VNFs composing such (sub-) slices, so as to fulfill the target KPIs of all involved services, and (iv) to instantiate the VNFs that have to be deployed *ex novo*, and allocate a suitable amount of resources (e.g., CPU, memory, routers) for their processing and interconnection.

Such a process, also referred to as *service orchestration* or *service embedding*, entails multiple, inter-dependent decisions including VNF placement and virtual machine (VM) provisioning. In making such decisions, the orchestrator needs to avoid two main sources of inefficiency. The first is due to placing VNFs with different delay constraints on the same VM: in such a case, the most stringent delay constraint is effectively maintained also for the least demanding VNF, which implies that some of the processing allocated to handle the other VNF(s) is not minimal. Such additional capacity can be seen as derived from latency *dissimilarity* of *co-located* VNFs and it is removed when all those placed in a VM have exactly the same delay constraint. The second inefficiency is due to assigning ("packing") VNFs in the VMs in a sub-optimal manner, that is, using more than the minimal number of VMs required.

Several works have addressed these aspects, including [1]–[7], by casting them into MILPs and proposing effective heuristics. Most of the existing studies, however, operate *offline*, i.e., assume that the set of service requests to deploy is known in advance and make the best (e.g., cost-effective) decisions for their instantiation. Importantly, our work is able to capture the effect of time-varying traffic conditions, i.e., with service instances exhibiting finite lifetime and time-varying traffic load and the flexible relationship between the computational resources assigned to a VNF and the resulting service times.

Further, unlike prior art [3], [4], [6], [7], our work accounts for the fact that next-generation networks are made of segments (e.g., cloud and edge) with different computational capability, latency, and cost: in this scenario, it is important that each service is supported by the segment with the most appropriate cost-performance trade-off. Furthermore, unlike state-of-the-art works like [8], [9], our system model and algorithms can account for the dynamic nature of the load networks have to serve. Such aspects are especially critical, as they influence a solution's ability to cope with daily and weekly traffic patterns and the network topology.

In this paper, we make the following main contributions:

- we first develop a model that captures the main aspects of the NFV ecosystem and accounts for a *time varying*

G. Einziger and G Scalosub are with the Ben Gurion University of the Negev, Israel. C. F. Chiasserini is with Politecnico di Torino, Italy. F. Malandrino and C. F. Chiasserini are with CNR-IEIIT and CNIT, Italy.

TABLE I
TABLE OF NOTATION

Symbol	Description
$\mathcal{S}$, $\mathcal{V}$	set of services and VNFs (resp.)
$\mathcal{V}_r^s$	set of VNFs of request r for service s
a_r, τ_r	arrival and duration (resp.) time of service instance request r
λ_r	load of service instance request r
D_r^s	request r delay constraint
$\mathcal{G}$	graph of layered topology of datacenters (nodes)
ℓ	layer of a node (distance from leaf)
b	VM running in a node
$\bar{\mu}$, μ	maximum and actual (resp.) computing capability of a VM
θ_v	computing complexity of VNF v
l_r	leaf node where request r arrives
$\Lambda(b)$	overall load on VM b
d_ℓ	forwarding latency from a leaf to a node in layer ℓ
$M_{r,v}$	minimum processing latency of running (r, v) alone in a VM
D_r^v	fair delay allocation of job (r, v)
κ_f^ℓ	fixed cost of a VM at level ℓ
κ_p^ℓ	proportional cost of a VM at level ℓ
σ	sequence of requests

system load, as well as for the fact that there are different segments composing the network;

- we formulate the problem of service embedding taking the end-to-end latency as the main service KPI, and develop an algorithm, named REShare, which addresses all issues listed above – network segment identification, VNF placement/reuse, and resource scaling – and, importantly, handles a time-varying service requests load;
- we prove that REShareha low, namely, quadratic, computational complexity and is an *asymptotic 2-competitive algorithm* under a non-decreasing load;
- finally, using real-world load traces, we show that the cost of deploying and running services under REShare is much lower than under state-of-the-art solutions. Also, REShare can swiftly adapt to time-varying conditions, attaining excellent performance as the system load evolves.

The rest of the paper is organized as follows. Sec. II introduces the system model, while Sec. III outlines our approach and main results. Sec. IV presents a heuristic strategy, which is our cornerstone for building the REShare algorithm, and analyzes its competitive ratio. REShare, along with its competitive analysis, is introduced in Sec. V, while its performance is assessed in Sec. VI. After discussing the related work in Sec. VII, we conclude the paper in Sec. VIII.

II. System Model

Our system model seeks to concisely represent the main features of next-generation network architectures, namely, their layered structure [10] and heterogeneity. Indeed, as reported in [11]–[13] for datacenters and in [14] for 5G networks, many relevant types of networks are organized in *layers*, with nodes at the same layer having similar features in terms of latency and cost; however, the features of different layers can be vastly different. In other words, any differences between nodes of the same layer are negligible when compared to those between nodes of different layers. By capturing such aspects, our model serves as the first step towards meeting the challenge they represent. Furthermore, we consider the NFV Orchestrator (NFVO), specified within the well-known NFV MANO (Management and Orchestration) architecture [15], as the decision-making entity. In both ETSI standards [15] and real-world 5G deployments [14], the NFVO has access to plentiful information concerning such aspects as: (i) the target KPIs of each service, e.g., its target end-to-end delay; (ii) the time for which each service instance shall be active; (iii) the VNFs composing the service; (iv) their computational requirements, which in turn determine the processing delay.

More specifically, the processing delay associated with a service is determined by (i) the available computational capabilities; (ii) the service computational complexity; and (iii) the service demand. The computational capabilities available to each service is one of our decision variables, as set forth next. As for the computational complexity of a given service, it is routinely obtained by profiling the service as it runs, hence, it can be considered a known quantity. Finally, the service demand can be either known *a priori* (e.g., in smart-factory applications) or reliably predicted [16], [17], as better detailed in Sec. VII.

We leverage these notions throughout our system model and algorithms, as set forth next. The most relevant notation we use is summarized in Tab. I.

Services. Let $\mathcal{S}$ be the set of services that verticals may request and $\mathcal{V}$ the set of all possible VNFs they may use. Requests for the deployment of service instances arrive sequentially at the system, and each new request r for an instance of service s requires a set[1] of connected VNFs $\mathcal{V}_r^s \subseteq \mathcal{V}$. Such VNFs may either be freshly deployed or reuse existing VNF instances provided that isolation requirements are met.

Verticals and network operators commit to service level agreements (SLAs), detailing the cost and the quantity of the resources each vertical can use. Thus, the requests received at the NFVO do not exceed the resource budget for which the vertical is paying; i.e., there are always enough resources to deploy a service meeting the target KPIs. Thus, our problem is not whether or not to *admit* a service request, but how to *embed* it in the most efficient way.

Network system. Most real-world networks connecting datacenters exhibit a layered topology [11], [12], e.g., trees, fat trees, as well as SlimFly and DragonFly [13] (an example of such a layered topology appears in Fig. 1). Let $\mathcal{G} = \langle \mathcal{I}, \mathcal{E} \rangle$ be a graph modelling such a layered topology, with vertices, $i \in \mathcal{I}$, representing datacenters (also referred to as *nodes*), and edges, $e \in \mathcal{E}$, representing communication links between datacenters. Each node can host a large [18] number of virtual machines (VMs). Each VM implements one VNF instance [14], and the same VNF instance can serve traffic belonging to one or more services.

Layers are numbered as follows: leaf nodes are at layer 0, and a generic node is at *layer* $\ell > 0$ if its distance in links from the closest leaf is ℓ. VMs are characterized by their *computational capabilities*, i.e., resources at their disposal. Intuitively, the more capable a VM is, the quicker it will be able to perform a given task. Specifically, a VM b running in some generic datacenter in node i has maximum speed

[1]We denote the set of VNFs of a service instance request (and the target latency introduced next) by $\mathcal{V}_r^s$ (D_r^s), even if it depends only on s.

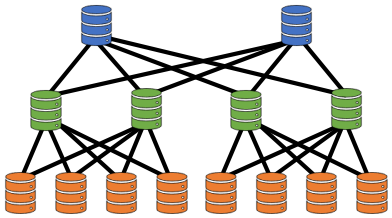

Fig. 1. Example of a layered topology, featuring three layers.

(i.e., computing capability) $\bar{\mu}$ and can host a single VNF v. We denote with $\mu_b \leq \bar{\mu}$ the amount of computing resources VM b is using, and with θ_v the *computing complexity of VNF* v. Intuitively, computational complexity expresses how difficult it is to run a certain VNF. If we fix the amount of computational capabilities, then the processing time required by a given task is proportional to its computational complexity. More formally, computational complexity is defined as the number of computing units[2] required by VNF v to handle a unit of traffic: the higher θ_v, the more complex the associated VNF, and the more computational resources will be required to run it.

For ease of presentation, we treat the computing capability μ as a scalar quantity, i.e., a number; therefore, μ is able to represent one type of system resources, e.g., CPU. Additional types of resources, like memory or storage, could be accounted for by converting both μ and the service requirements to multi-dimensional variables, i.e., vectors. Our approach would work unmodified; however, so doing would entail significant additional notation complexity.

A request r is associated with a tuple $\langle \mathcal{V}_r^s, a_r, \tau_r, D_r^s, \lambda_r, l_r \rangle$, where l_r specifies the leaf node at which the request arrives, a_r and τ_r are, respectively, its arrival time and duration, λ_r is the traffic load, and D_r^s is the latency requirement. Having a finite lifetime for requests allows us to model time-varying network load. Specifically, as demonstrated in Sec. VI, fluctuations in the network load can be reproduced by adding, removing, or replacing service requests. Notice that the duration (i.e., lifetime) of a service is a distinct quantity from the time it takes to process an individual packet (or query) belonging to the service, and the quantities often differ by orders of magnitude. As an example, a smart factory service like the one discussed in Sec. VI-A may be active for as long as a certain batch has to be produced, i.e., its service duration can be of minutes or days; in the other hand, individual messages pertaining to robot actions are processed in milliseconds.

For each VNF $v \in \mathcal{V}_r^s$, we define (r, v) as the *job* of running VNF v for request r; each job (r, v) has to be assigned to a VM b hosting v, running on some node. Clearly, the jobs associated with a request arrive all upon the request's arrival (i.e., at time a_r), and depart after the request's duration τ_r has elapsed (i.e., at time $a_r + \tau_r$), when all jobs (r, v) are removed and the resources they consumed are freed. Finally, $\Lambda(b)$ denotes the overall traffic load of the jobs assigned to VM b.

[2]For simplicity, $\theta_v \leq 1$ is normalized to the maximum VNF complexity.

Latency. The total latency incurred by request r is given by the sum of the traffic forwarding latency and the processing latency. To determine the former, we consider that routes between any two nodes in the topology have been pre-computed. We denote with d_ℓ the traffic forwarding latency from a leaf node to a node at layer ℓ; owing to the layered structure of the network topology (e.g., edge, aggregation, and cloud layers), we have $d_{\ell+1} > d_\ell$, reflecting the fact that farther-away layers take longer to reach than close-by ones. We further consider the forwarding latency incurred by request r, where each job (r, v) is deployed at some level ℓ_r^v, as the *maximum* forwarding latency over all levels hosting jobs of r. The maximum forwarding latency captures the latency incurred by reaching the highest level, which hosts jobs corresponding to the request. Such a latency is the dominating component of the overall forwarding latency incurred by the job; indeed, as better detailed in Sec. IV, the large number of VMs available in each datacenter makes placing all VNFs of a service in the same node (albeit on different VMs) the most convenient option, hence, inter-VNF forwarding times are not significant.

As for the processing latency, we model each VM as an $M/M/1$ queue. Queuing models are commonly used in the literature to represent VMs in edge and cloud scenarios [1], [19]–[22], with Markovian service [1], [20], [22] and arrival [1], [19], [20], [22] times being considered in most cases; the validity of such modeling assumption has also been confirmed through simulation-based validation [21]. Using a queueing model allows us to model how VNFs may be assigned different quantities of resources, e.g., CPU, and how such an amount impacts their processing time. This is in contrast with existing works, e.g., [3], [4], [6], [7], where VNF requirements are constant over time and no resource allocation decision is possible.

As per [23], the processing latency incurred on VM b running VNF v at speed μ_b is $\frac{1}{\mu_b - \theta_v \Lambda(b)}$. While VMs get assigned virtual cores in practice, it is fair to assume that μ_b's take on real values: indeed, given that each node corresponds to a datacenter, it contains a very large number of physical cores, and each physical core can accommodate a high number (e.g., up to 32) of virtual cores. Importantly, a queuing model allows us to capture the flexible relation between the computational resources assigned to a VNF and the resulting processing times, i.e., VNFs process requests quicker if they have more resources. For stability, we must have $\Lambda(b) < \mu_b / \theta_v$.

A deployment is said to be *feasible* for request r if the sum of the traffic forwarding latency and processing latency over all $v \in \mathcal{V}_r^s$, with the latter computed considering the maximum VM capability $\bar{\mu}$ for all VNFs, does not exceed target delay D_r^s. Also, for every r, let $\ell^*(r)$ be the highest layer for which running all $v \in \mathcal{V}_r^s$ on VMs in a node at that layer is a feasible deployment. Next, we define the *fair per-job delay allocation* over the jobs (r, v) as the per-job processing latency budget, which will serve as a guide for placing request's jobs, allocating computing resources to them, and performing VMs sharing across distinct jobs.

The intuition behind the fair delay allocation is to give more resources to more complex VNFs, i.e., those with larger

computational requirements θ_v. Such an allocation is *fair*, in that no VNF contributes in a disproportionate manner to the total processing time. Specifically, since a deployment of request r is feasible if the total processing latency of its jobs is at most $D_r^s - d_{\ell^*(r)}$, then we distribute this latency budget over the distinct jobs (r, v) according to their *relative* processing requirements. Namely, for each (r, v), the fair per-job delay allocation is: $D_r^v = \frac{M_{r,v}}{\sum_{u \in \mathcal{V}_r^s} M_{r,u}}(D_r^s - d_{\ell^*(r)})$, where $M_{r,v} = \frac{1}{\bar{\mu} - \theta_v \lambda_r}$ is the latency incurred by running v using the maximum VM capability $\bar{\mu}$, and $\frac{M_{r,v}}{\sum_{u \in \mathcal{V}_r^s} M_{r,u}}$ accounts for the relative time complexity of $v \in \mathcal{V}_r^s$. It follows that a job (r, v) corresponding to a VNF v with a larger value of θ_v will be assigned a larger value of D_r^v.

Consider now request r such that job (r, v) is deployed on a node at layer $\ell_r^v \leq \ell^*(r)$, while satisfying the fair per-job delay allocation. The overall processing latency of the request is at most:

$$\sum_{v \in \mathcal{V}_r^s} D_r^v = \sum_{v \in \mathcal{V}_r^s} \frac{M_{r,v}}{\sum_{u \in \mathcal{V}_r^s} M_{r,u}}(D_r^s - d_{\ell^*(r)}) \tag{1}$$

$$= D_r^s - d_{\ell^*(r)}. \tag{2}$$

By the definition of $\ell^*(r)$ and the fact that forwarding latency is monotone with the node's level, we have that the overall latency of deploying request r is at most D_r^s, and thus feasible. In the following, we will consider feasible service deployments that satisfy such fair per-job delay allocation, and we will compare our solution to an optimal solution that must also satisfy such fair per-job delay bounds.

Cost. Running a VM b at a node of layer $\ell(b)$ implies a *fixed* cost, $\kappa_f^{\ell(b)}$. The fixed cost reflects the fact that unused VMs still entail additional work for the hypervisor and keep resources allocated, both of which consume power [24], [25], hence, money. Furthermore, any VM b running on a node at layer $\ell(b)$ at speed μ_b incurs cost $\kappa_p^{\ell(b)}\mu_b$, which is proportional to the computing resources consumed by b. Our model accounts for the fact that running VMs at a higher layer (closer to the cloud) is cheaper [26] than running VMs at lower layers (closer to the edge), i.e., $\kappa_f^{\ell+1} < \kappa_f^{\ell}$ and $\kappa_p^{\ell+1} < \kappa_p^{\ell}$. Cost parameters can also account for differently-owned and/or federated resources, e.g., in different administrative domains.

Denoting by σ the sequence of the requests handled by the system till the current time, we define $\phi(A, \sigma)$ as the overall *instantaneous* cost of the deployment decisions made by an algorithm A, i.e., the sum of fixed and proportional costs of all VMs currently used. $\phi(A)$, instead, refers to the instantaneous cost of algorithm A at the end of the whole request arrival/departure process.

A. Problem formulation

The main decisions we have to make are (i) whether VM b should be used for job (r, v), expressed through binary variables $y(r, b, v) \in \{0, 1\}$, and (ii) how much computational resources to assign to each VM, expressed through the real variable μ_b. Importantly, y-variables also express whether or not a certain VM b is active; specifically, b is active – hence, the corresponding fixed cost is incurred – if and only if it is used by at least one job, i.e., $\sum_{(r,v)} y(r, b, v) \geq 1$. Also notice that, in our system model, VMs do not migrate, hence, the values of the y-variables do not change during the lifetime of a job.

Our goal is to minimize the total cost, i.e.,

$$\min_{y, \mu_b} \sum_b \left(\kappa_f^{\ell(b)} \max_{(r,v)} y(r, b, v) + \kappa_p^{\ell(b)} \mu_b \right). \tag{3}$$

The two terms in (3) correspond to the fixed and proportional cost, respectively. The constraints we have to satisfy are that all target delays are met: for each job (r, v), if b is the VM serving it, i.e., $y(r, b, v) = 1$, then we wish to impose $\frac{1}{\mu_b - \theta_v \Lambda(b)} \leq D_r^v$. This is captured by requiring

$$D_r^v \cdot y(r, b, v)\,(\mu_b - \theta_v \Lambda(b)) \geq y(r, b, v) \qquad \forall (r, v), \tag{4}$$

where $\Lambda(b) = \sum_{r'} y(r', b, v)\lambda_{r'}$. We further impose that

$$\sum_b y(r, b, v) \geq 1, \qquad \forall (r, v), \tag{5}$$

$$\sum_v \max_r y(r, b, v) \leq 1, \qquad \forall b, \tag{6}$$

$$\mu_b \leq \bar{\mu}, \qquad \forall b, \tag{7}$$

where Eq. (5) ensures that every job is actually deployed, Eq. (6) mandates that each VM runs a single VNF, and Eq. (7) ensures that the capacity constraint of each VM is met. Since the problem is combinatorial in nature and non-linear, and the formulation is non-convex, we adopt a heuristic approach in designing an efficient and effective algorithm for solving the problem.

III. Approach and Main Results

We now describe the approach we use in designing our algorithm, REShare, which performs feasible VNF placement with *time-varying service demand*. REShare aims at minimizing the overall cost, and it is shown to be *asymptotically 2-competitive* when requests duration, τ_r, is unlimited.

REShare tackles the sources of inefficiency described in Sec. I, as follows:

- to reduce the number of requests served before their deadline (and the resulting waste or resources), REShare seeks to minimize the *dissimilarity* between jobs that are co-located in the same VM;
- to reduce the bin-packing suboptimality, REShare leverages efficient, state-of-the-art bin-packing algorithms.

Our algorithm carefully manages these challenges according to the overall system load. Intuitively, we pack together diverse services at times of low system load to avoid creating many underutilized VMs. For high system loads, instead, we want to utilize individual VMs better and allow for minimal dissimilarity.

Our solution is based on a carefully designed algorithm, constant-REShare(c-REShare), with parameter ε, described in Sec. IV. The value of ε governs the amount of dissimilarity allowed; smaller ε values imply less dissimilarity. c-REShare(ε) places all jobs corresponding to a request in a single node at the highest level for which the placement is feasible (even if

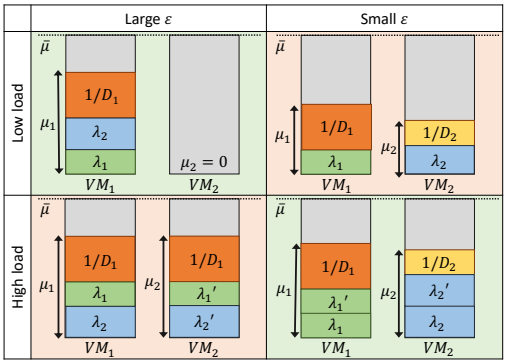

Fig. 2. Example of the c-REShare approach: consider two types of services (1 and 2), each composed of the same, single VNF. The target latency is D_1 and D_2 (resp.), with $D_1 < D_2$. As detailed in Sec. IV, the capacity required at each VM is given by (i) the total load it must serve (green and blue blocks), plus (ii) the inverse of *the shortest delay* it must guarantee (orange and yellow boxes). Under low load (one request per service), a large ε (top left) entails that the two requests can share the same VM and, considering the M/M/1 modeling, consume $\mu_1 = \lambda_1 + \lambda_2 + 1/\min(D_1, D_2)$. Instead, a small ε (top right) requires two VMs and a larger total capacity. For high load (two requests per service), a large ε (bottom left) means that services can share the same VMs, but require a high capacity as the most stringent target latency must be met by both VMs. A small ε (bottom right) yields a cheaper deployment instead.

it requires starting a new VM for every job). c-REShare(ε) is shown to be *asymptotically* $(2+\varepsilon)$*-competitive* under a non-decreasing load.

REShare uses c-REShare(ε) with dynamically varying values of ε, depending on the overall system load. When the system load is low, bin-packing suboptimality is the dominant cost factor, and REShare uses a larger value for ε. When system load increases, bin-packing suboptimality is naturally reduced since any solution must use many VMs for supporting the load. In this case, REShare favors reducing dissimilarity by using a smaller value for ε. Fig. 2 provides an example of these aspects. One of the main novel elements of our approach is such an ability to switch between different decision-making strategies (expressed through different values of ε) as the network load varies. Indeed, recent works [3], [4] leverage a forecast of future requests for the next few hours; [6] accounts for the network load evolution but always follows the same strategy, i.e., the same trade-off between resource consumption and overhead. Finally, [7] does not adapt its decision strategy to the load and deploys multiple replicas of each VNF chain to achieve robustness.

We decide when to transition between ε values by maintaining a lower bound on the cost of an optimal solution (which is closely correlated with the overall system load). This approach also allows us to prove that REShare is asymptotically 2-competitive when requests duration, τ_r, is unlimited.

Notice that assuming infinite request duration is only required for our analysis. In practice, REShare is inherently designed to deal with time-varying load conditions where requests arrive and leave the system. We show the effectiveness of our approach through an extensive simulation study, presented in Sec. VI, which provides further insight into the dynamic behavior of REShare and the reasoning underlying our algorithmic approach.

A. An AWS-based experiment

Before describing our c-REShare approach, we perform a simple real-world experiment to demonstrate the first, and perhaps less intuitive, of the suboptimality sources that we discuss, namely, serving requests with different deadlines within the same node. We consider two types of requests:

- a MAFFT protein alignment task [27] whose target delay is 80 s;
- a WAV-to-FLAC audio conversion [28] whose target delay is 50 s.

We further consider two types of VMs available on Amazon AWS, namely, `m1.small` and `m2.xlarge`. For both VMs, we select the Ubuntu operating system, and run the benchmarks via the `phoronix` command. When ran alone, the MAFFT benchmark takes 37 s on an `m2.xlarge` machine, and 137 s on an `m1.small` one; the FLAC benchmark takes 11 s on an `m2.xlarge`, and 48 s on an `m1.small`.

A total of three requests arrive at very short interval from each other, in the following order: first a request for the MAFFT task (`mafft_1`), then one for the FLAC task (`flac_1`), and finally a second MAFFT request (`mafft_2`). Recall that we take an online approach, i.e., we embed all requests as soon as they arrive.

The possible decisions are summarized in Fig. 3. Clearly, when request `mafft_1` arrives (step 1), we must create a new VM. Since an `m1.small` VM would be too slow, we must open an `m2.xlarge` VM, called `xlarge_1` (step 2). After job `flac_1` arrives (step 3), we can make two decisions: re-use existing VM `xlarge_1`, or open a new VM `small_1` of type `m1.small`.

Re-using `xlarge_1` (step 4a) would minimize both the number of VMs and the cost *so far*, however, things would be different after the arrival of the third job (step 5): using `xlarge_1` (step 6a) would result in a violation of the target KPIs, hence, we need to open a new VM. Furthermore, such a new VM can only be of type `m2.xlarge` (step 6b), resulting in a higher cost of 1 USD/hour[3].

Let us assume instead that we create a new VM for the newly-arrived `flac_1` job, on the grounds that its target delay and complexity are vastly different from the ones of job `mafft_1`; importantly, a cheaper VM of type `m1.small` would suffice. Performing this counter-intuitive action allows us to serve request `flac_2` on the existing VM `xlarge_1` (step 6c), resulting in a total lower cost of 0.585 USD/hour.

In summary, avoiding serving requests with overly-different values of target delay within the same VM results in lower global costs. As confirmed in Tab. II, it also results in requests served closer to their deadline, hence, intuitively, in less wasted computational capabilities.

IV. The c-REShare(ε) Strategy

This section describes and analyzes the constant-REShare (c-REShare) strategy, with a fixed parameter ε.

[3]All prices refer to the US-east availability zone.

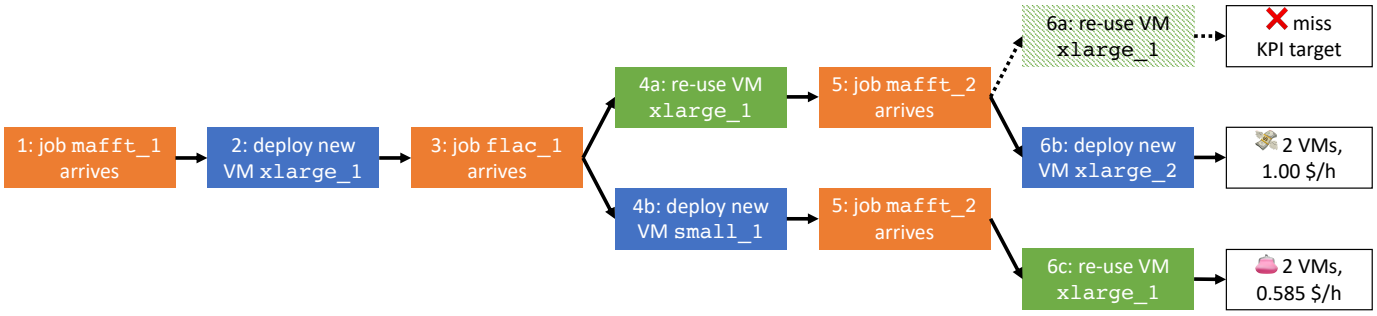

Fig. 3. AWS-based experiment: summary of the possible decisions and their outcome.

TABLE II
AWS-BASED EXPERIMENT: SERVICE TIME AND COST FOR THE DIFFERENT OPTIONS REPORTED IN FIG. 3

case	service time [s]	cost [USD/hour]
6a	mafft_1: 85 mafft_2: 85 flac_1: 33	0.5
6b	mafft_1: 48 mafft_2: 22 flac_1: 37	1
6c	mafft_1: 74 mafft_2: 74 flac_1: 45	0.585

A. Algorithm description

The details of c-REShare(ε) are presented in Alg. 1, which takes ε, determining the level of VNF (hence, VM) sharing among jobs, as an input parameter. Given a request r, first c-REShare(ε) identifies layer $\ell^*(r)$ (Line 3), which is the highest layer where the requested service instance can be deployed without violating the service KPI targets and keeping the fair per-job delay allocation (see Sec. II). We remark that such a layer is selected because VNF deployment is cheaper at higher layers and that this layer can be found by conducting a binary search, i.e., in time that is logarithmic in the number of layers. Furthermore, c-REShare(ε) deploys all the service VNFs in the same node to minimize the traffic forwarding latency and maximize the computation latency budget. Recall that, as per our system assumptions, it is *always* possible to place in the same node all VNFs composing a service. Furthermore, whenever doing so is possible, it is also beneficial, as it avoids incurring inter-node forwarding latency.

The specific node i^* at layer $\ell^*(r)$ where the request traffic should be processed is chosen as the one providing the best load balancing (Line 4), i.e., the one that currently has the lowest load. However, it is important to highlight that the choice of i^* does not affect the worst-case performance of the algorithm and that we can use alternative criteria (e.g., choosing the most-recently-used or most-loaded server) with similar guarantees.

Assigning jobs (r, v) to VMs can now be done *independently* over the VNFs, as long as the fair per-job delay allocation, D_r^v, $v \in \mathcal{V}_r^s$, is met. Placing each job in a new or existing VM running v poses the following dilemma:

- on the one hand, we would like to *share* VMs as much as possible, to save on fixed costs;
- on the other, sharing the same VM among jobs with different delay allocation results in wasted capacity (as discussed earlier and exemplified in Fig. 2), and, thus, in higher proportional costs.

Algorithm 1 c-REShare(ε)

1: **for** any arrival/departure of request r **do**
2: **if** r is arriving **then**
3: $\ell^*(r) \leftarrow \max\{\ell | \text{fair delay allocation is met at } \ell\}$ ▷ choose highest possible layer
4: $i^* \leftarrow \text{Load\_balancing}(G, \ell^*(r))$ ▷ select node
5: **for** each $v \in \mathcal{V}_r^s$ **do**
6: $j^* \leftarrow \left\lfloor \log_{(1+\varepsilon)} D_r^v \right\rfloor$ ▷ determine latency range
7: VM_assignment$((r, v), (i^*, v, j^*)$-AP) ▷ assign job to a VM
8: **else** ▷ r is departing
9: **for** each $v \in \mathcal{V}_r^s$ **do**
10: VM_deassignment$((r, v))$ ▷ deallocate the job

The parameter ε describes the tradeoff underlying such decisions. Specifically, let $\lambda_{\min} = \inf_r \{\lambda_r\}$, where we assume $\lambda_{\min} > 0$ and is known in advance; then we define a series of non-overlapping latency ranges L_j, where $L_j = \left(\frac{1}{\bar{\mu}-\lambda_{\min}(1+\varepsilon)^j}, \frac{1}{\bar{\mu}-\lambda_{\min}(1+\varepsilon)^{j+1}}\right]$, for $j \geq 1$, and $L_0 = \left[\frac{1}{\bar{\mu}-\lambda_{\min}}, \frac{1}{\bar{\mu}-\lambda_{\min}(1+\varepsilon)}\right]$. Clearly, we need to impose that $\lambda_{\min}(1+\varepsilon)^{j+1} < \bar{\mu}$, $\forall j$; then, the number of latency ranges, J_ε, satisfies $J_\varepsilon < \log_{(1+\varepsilon)} \frac{\bar{\mu}}{\lambda_{\min}} - 1$. c-REShare($\varepsilon$) only shares the same VM among jobs within the same latency range. Thus, a larger value of ε results in wider ranges and jobs with more diverse fair delay allocations sharing the same VM. On the contrary, a smaller value of ε implies narrower ranges and more *similar* jobs within each range.

Given i^*, the objective is to assign each VNF $v \in \mathcal{V}_r^s$ to a suitable VM in i^*, such that the fair delay allocations are satisfied for all jobs on that VM (Line 6). In particular, given a VNF v, we consider the *VM Assignment Problem*, hereinafter referred to as (i, v, j)-AP, which assigns a job (r, v) to a VM in node i, while ensuring that $D_r^v \in L_j$ (with $j = \lfloor \log_{(1+\varepsilon)} D_r^v \rfloor$). Since we only share VMs among jobs in the same range, it is possible to consider separate assignment problems for different values of j.

Algorithm 2 VM_assignment($(r,v),(i,v,j)$-AP)

1: viable_VMs $\leftarrow \emptyset$
2: **for** every VM b in i hosting jobs (r',v) in L_j **do**
3: **if** $\frac{1}{\bar{\mu}-\theta_v(\Lambda(b)+\lambda_r)} > D^v_r$ **then** ▷ check feasibility for the new job on b
4: **continue** ▷ continue to next VM
5: **for** every $(r',v) \in b$ **do** ▷ jobs already assigned to b
6: **if** $\frac{1}{\bar{\mu}-\theta_v(\Lambda(b)+\lambda_r)} > D^v_{r'}$ **then** ▷ check feasibility for jobs already on b
7: **continue** ▷ continue to next VM
8: viable_VMs $\leftarrow$ viable_VMs $\cup \{b\}$ ▷ reusable VMs
9: **if** viable_VMs $\neq \emptyset$ **then**
10: $b^* \leftarrow \arg\max_{b \in \text{viable\_VMs}} \Lambda(b)$ ▷ Best-fit
11: $\mu_{b^*} \leftarrow \theta_v[\Lambda(b)+\lambda(r)] + \frac{1}{\min_{(r',v)\in b} D^v_{r'}}$ ▷ adjust capability
12: **else**
13: $b^* \leftarrow$ **create new VM** in i
14: $\mu_{b^*} \leftarrow \theta_v \lambda_r + \frac{1}{D^v_r}$ ▷ adjust capability
15: **place** (r,v) in b^*

Algorithm 3 VM_deassignment((r,v))

1: $i \leftarrow$ node running request r
2: $b \leftarrow$ VM on node i running job (r,v)
3: remove (r,v) from b
4: $\mu_b \leftarrow \theta_v \Lambda(b) + \frac{1}{\min_{(r',v)\in b} D^v_{r'}}$ ▷ adjust capability

The VM_assignment procedure for solving (i,v,j)-AP is presented in Alg. 2. It begins by looking for VMs whose capacity could be expanded to accommodate the additional load of job (r,v), while honoring its delay allocation D^v_r (Line 3) as well as that of previously allocated jobs (Lines 5-7). It then places such VMs in set viable_VMs (Line 8). If the set is not empty, in Line 10 the viable VM with the least free capacity is selected, in a Best-fit [29] fashion. If there are no viable VMs, a new VM is instantiated in Line 13. In either case, job (r,v) is assigned to VM b^* (Line 15). In Line 11 (and similarly in Line 14), the capacity of b^* is adjusted to guarantee that the fair delay allocations for all jobs assigned to b^* are met. Through simple manipulation of the M/M/1 delay expression, this corresponds to setting $\mu_{b^*} = \theta_v[\Lambda(b)+\lambda(r)] + \frac{1}{\min_{(r',v)\in b} D^v_{r'}}$. Line 8, and the fact that the delay allocation of job (r,v) can always be satisfied by placing it in a new VM, imply that it is always possible.

The departure of service instances is dealt with in Alg. 3, which removes all jobs of request r from the VMs hosting them and readjusts the VMs capability to release the resources while meeting the constraints of all remaining jobs.

B. Competitive ratio analysis

In this section, we compare the cost of c-REShare(ε) against that of an optimal solution, denoted by OPT, which is aware of the future sequence of service requests that arrive at the NFVO. We derive our worst-case performance guarantees for scenarios where each request has infinite duration. This analysis guides our algorithmic design and provides asymptotic performance guarantees when system load tends to infinity. However, as we show in later sections, our proposed solutions also provide significantly improved performance in cases where requests have a finite duration, and the system load increases and decreases in a time-varying manner. We first provide below an overview of our analysis.

High-level description of the analysis. At the heart of our analysis lies a load argument across distinct layers, where we compare the load handled by c-REShare(ε) at some layer to that handled by OPT. We consider a shadow solution, SHA(ε) (described in the sequel), which is allowed to produce a *fractional assignment*, and should also satisfy *relaxed delay constraints*. This solution runs *most* of its VMs at full capacity, and the cost of such VMs serves as a *lower bound on the cost of OPT*. We show that the cost induced by the deployment performed by c-REShare(ε) is comparable to that of this lower bound, and provide an additional bound on the *additive* cost of all remaining VMs that may be used by OPT. In what follows, we provide a detailed account of our analysis.

We first show that c-REShare(ε) performs feasible deployments for any service request.

Lemma 1. *If c-REShare(ε) assigns r to node i^* at layer $\ell^*(r)$, allocating VM capability μ_{b^*}, then such a deployment of the service instance is feasible.*

Proof. Consider r being placed at node i^*, located at layer $\ell^*(r)$. By the definition of Alg. 1, $\ell^*(r)$ is the highest layer where deploying all $v \in \mathcal{V}^s_r$ on new VMs in a node at layer ℓ is feasible. When considering Alg. 2, each job (r,v) is placed in one of the viable VMs at i^* where running the VM at maximum processing capability ensures a feasible solution (Lines 2–7). Since Alg. 2 uses the minimal capability that ensures feasibility (Lines 11–14), the result follows. □

Next, we compare the performance of c-REShare(ε) to the optimum. To bound the competitive ratio and leverage load rearrangement arguments in our proofs, we introduce an alternative request (hence, job) arrival/departure process, and an alternative placement strategy. Specifically, for any job (r,v) associated with latency range L_j, i.e., such that $D^v_r \in \left(\frac{1}{\bar{\mu}-\lambda_{\min}(1+\varepsilon)^j}, \frac{1}{\bar{\mu}-\lambda_{\min}(1+\varepsilon)^{j+1}}\right]$ (or, possibly, $D^v_r = \frac{1}{\bar{\mu}-\lambda_{\min}}$ in the case of $j=0$), we define a corresponding *top* job, $\overline{(r,v)}$, as equivalent to (r,v) but associated with delay constraint $\overline{D^v_r} = D_j = \frac{1}{\bar{\mu}-\lambda_{\min}(1+\varepsilon)^{j+1}} \geq D^v_r$. [4] Note that all top jobs falling in the same latency range L_j have the same delay constraint D_j.

A shadow strategy. We now introduce the *shadow fractional assignment (SHA(ε))*, a *fractional* placement strategy, whose cost will be used to determine a lower bound on the cost of OPT within the analysis of c-REShare(ε). Furthermore, in Sec. V we also use SHA(ε) explicitly within our algorithm REShare, which can handle time-varying load.

SHA(ε) uses the same ε value as c-REShare(ε), and operates as follows: (i) it handles the top job sequence $\overline{(r,v)}$; (ii) it

[4] Note that the load offered by $\overline{(r,v)}$ is the same as that of (r,v), and the difference is that the top job has a (possibly) more relaxed delay constraint.

never places in the same VM jobs associated with different L_j (hence, by (i) and the definition of top jobs, with different delay allocations); (iii) it places a job in a single node; (iv) within the chosen node, it can place fractions of a job load on different VMs (i.e., it works with a "fluidified" version of the jobs); (v) it generates the *optimal* placement that satisfies conditions (i)-(iv) above. In case of a request departure, SHA(ε) removes the load associated with the corresponding jobs and fractionally rearranges the remaining load; thus, the cost of the resulting placement is equivalent to that of re-running SHA(ε) on all the remaining requests.

SHA(ε) is used in two manners: (i) guiding the decisions of our *dynamic* algorithm, REShare (details are provided in Sec. V), and (ii) identifying a *lower bound on the cost of OPT*. We now briefly explain how the cost of SHA(ε) can be used to define a lower bound on the cost of OPT, where we refer to this property as the *relaxation property of SHA(ε)*. Note that by using the top sequence with a fractional assignment, for each VNF v, every latency value D_j of a top job, and every node i, all the VMs, save at most one, used by SHA(ε) in i work at full computing capacity $\bar{\mu}$ to handle jobs associated with v and D_j. Since such full VMs handle the load placed upon them in the most efficient manner (i.e., there are no inefficiencies due to placing jobs with different delay constraints on the same VM), this amount of load must also be handled by OPT. All other properties of SHA(ε) essentially relax the constraints imposed on OPT. It follows that the cost of running these full VMs serves as a lower bound on the cost of OPT.

We begin our analysis by showing that c-REShare(ε) places jobs in the highest layer possible.

Lemma 2. *If c-REShare(ε) places the jobs of request r in node i^* at layer $\ell^*(r)$, then OPT and SHA(ε) both place every job (r,v) in some node i' at some layer ℓ', with $\ell' \leq \ell^*(r)$.*

Proof. We start by showing that the claim holds for OPT. Assume, by contradiction, that OPT places some job (r,v) at node i' at layer $\ell' > \ell$. Then, the network latency of OPT is higher than that of c-REShare(ε) (since the traffic associated with r has to reach layer $\ell' > \ell$), i.e., $d_{\ell'} > d_\ell$. As for the processing latency yielded by the OPT placement, this is at least $\sum_{v \in V_r^s} M_{r,v}$. By construction, c-REShare(ε) ensures that node i^* is at the highest layer $\ell^* = \ell^*(r)$ for which $\sum_{v \in V_r^s} M_{r,v} \leq D_r^s - d_{\ell^*}$; this implies that the assignment made by OPT is not feasible, thus contradicting the initial assumption. By item (iii) in the definition of SHA, the above argument also holds for SHA(ε). □

In the remainder of this section, we consider that requests have an infinite duration, i.e., $\tau_r = \infty$, for all arriving requests r. Under this condition, we can bound the total amount of traffic load (over all possible service requests) that SHA(ε) may process at a different layer than the one selected by c-REShare(ε) for any given node used by c-REShare(ε), any VNF, and any latency range.

Lemma 3. *For every node i at layer ℓ, every VNF v, and latency range L_j, the overall load of VNF v, handled by c-REShare(ε) at i and associated with L_j, that is handled by SHA(ε) at a layer $\ell' \neq \ell$, is at most $n_{\ell'} \cdot \lambda_{\min}(1+\varepsilon)^{j+1}$, where $n_{\ell'}$ is the number of nodes at layer ℓ'.*

Proof. Assume by contradiction that there exist some VMs running VNF v at layer ℓ', associated with latency range L_j, that, according to SHA(ε), handle a load higher than $n_{\ell'} \cdot \lambda_{\min}(1+\varepsilon)^{j+1}$, while c-REShare($\varepsilon$) handles that load at node i at layer $\ell \neq \ell'$. By Lemma 2, we have that $\ell' < \ell$. By the pigeonhole principle [30], there exists at least one node at a layer $\ell' < \ell$, such that there exists a total load of at least $\lambda_{\min}(1+\varepsilon)^{j+1}$ corresponding to L_j, that is handled by c-REShare(ε) in node i at layer ℓ, but is handled by SHA(ε) in node i' at layer ℓ'. Without loss of generality, we assume that there exists a *full* VM in node i' at layer ℓ' that, according to SHA, processes $\lambda_{\min}(1+\varepsilon)^{j+1}$ traffic with latency range L_j. This assumption is fair since SHA(ε) applies a fractional load assignment and places in the same VM only jobs associated with the same latency range, while the delay constraints of all jobs in that latency range are the same in the top sequence.

Consider now an alternative solution, SHA(ε)$'$, which is identical to SHA(ε), except for having this entire VM run at some node i'' at layer $\ell'+1$. First, note that this produces a feasible shadow fractional assignment for the workload handled by SHA(ε) and does not require increasing the processing speed of the VM at layer $\ell'+1$. Indeed, since c-REShare(ε) places these jobs at layer $\ell'+1$ or higher, the per-job latency constraint is satisfied for the solution of SHA(ε)$'$ and, by definition of L_j, SHA(ε) already processes $\lambda_{\min}(1+\varepsilon)^{j+1}$ traffic at i' using the maximum VM computing capability. Second, the cost of SHA(ε)$'$ is strictly smaller than the cost of SHA(ε) since, at layer $\ell'+1$, both the proportional cost and the fixed cost are smaller than at layer ℓ'. This contradicts the optimality of SHA, thus completing the proof. □

Next, for every VNF v, every node i, and every latency range L_j, let $\Lambda_{i,j}^v$ be the total load due to all jobs (r,v) that are handled by c-REShare(ε) in node i at layer ℓ. We next provide an upper bound on the maximum number of VMs that c-REShare(ε) requires to handle such load, which is then used to prove the competitive ratio of c-REShare(ε).

Lemma 4. *Given VNF v, node i, and latency range L_j, the number of VMs used by c-REShare(ε) to handle workload $\Lambda_{i,j}^v$ is at most $\frac{2\Lambda_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^j} + 1$.*

Proof. First, note that the delay constraint D_r^v of every job (r,v) contributing to $\Lambda_{i,j}^v$ is at least $\frac{1}{\bar{\mu} - \lambda_{\min}(1+\varepsilon)^j}$. It follows that we can pack a load of at least $\lambda_{\min}(1+\varepsilon)^j$, while satisfying the delay constraints of the jobs assigned to the VM. We therefore view this latter quantity as the *lower bound on the size* of the VMs in the (i,v,j)-AP.

The overall load on each VM b of (i,v,j)-AP, except for maybe one, is at least $\frac{\lambda_{\min}(1+\varepsilon)^j}{2}$. This argument follows from the fact that if there were two VMs with load strictly less than $\frac{\lambda_{\min}(1+\varepsilon)^j}{2}$, combining their loads on a single VM would have resulted in a VM with an overall load of at most $\lambda_{\min}(1+\varepsilon)^j$. This assignment is still feasible and incurs a lower placement cost. Thus, running such a VM at speed $\bar{\mu}$ would result in every job (r,v) assigned to the VM experiencing a latency

at most $\frac{1}{\bar{\mu}-\lambda_{\min}(1+\varepsilon)^j} \leq D_r^v$, where the inequality follows from the definition of L_j. This contradicts the definition of VM_assignment (Alg. 2), which assigns jobs to already open VMs if the overall load on the VM would still result in a feasible solution (for some speed no larger than $\bar{\mu}$). Hence, the overall number of VMs used by c-REShare(ε) for handling $\Lambda_{i,j}^v$ is at most $\left\lceil \frac{2\cdot\Lambda_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^j} \right\rceil \leq \frac{2\cdot\Lambda_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^j} + 1$, which proves the thesis. □

Theorem 1. *c-REShare(ε) is a $2(1+\varepsilon)$-asymptotic competitive algorithm when service requests have unlimited duration.*

Proof. Consider $\Lambda_{i,j}^v$; this can be seen as the sum of loads of two types of jobs: type 1, corresponding to jobs that are handled by SHA(ε) at some node i' at layer ℓ, with an overall load $\tilde{\Lambda}_{i,j}^v$, and type 2, corresponding to jobs that are handled by SHA(ε) at some node at a layer other than ℓ, with an overall load of $\bar{\Lambda}_{i,j}^v = \Lambda_{i,j}^v - \tilde{\Lambda}_{i,j}^v$. We denote the amount of resources used by c-REShare(ε) for handling $\Lambda_{i,j}^v$ by c-REShare$_{i,j}^v$, and that used by SHA(ε) for handling jobs contributing to $\bar{\Lambda}_{i,j}^v$ by SHA$_{i,j}^v$.

First, by Lemma 4, we have that the overall number of VMs used by c-REShare(ε) for handling load $\Lambda_{i,j}^v$ is at most

$$\begin{aligned}\frac{2\Lambda_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^j} + 1 &= \frac{2\tilde{\Lambda}_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^j} + \frac{2\bar{\Lambda}_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^j} + 1 \\ &\leq \frac{2\tilde{\Lambda}_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^j} + \frac{2n\lambda_{\min}(1+\varepsilon)^{j+1}}{\lambda_{\min}(1+\varepsilon)^j} + 1 \\ &= \frac{2\tilde{\Lambda}_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^j} + 2n(1+\varepsilon) + 1\,,\end{aligned}$$

where the inequality follows from Lemma 3. If we let $\kappa^\ell = \kappa_f^\ell + \kappa_p^\ell \bar{\mu}$ denote the cost of running a single VM at maximum speed in node i at layer ℓ, then

$$\phi(\text{c-REShare}_{i,j}^v) \leq \frac{2\tilde{\Lambda}_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^j}\kappa^\ell + [2n(1+\varepsilon)+1]\kappa^\ell\,. \quad (8)$$

We now derive a lower bound on the cost of the VMs used by SHA(ε) for handling jobs contributing to $\tilde{\Lambda}_{i,j}^v$, which serves as a lower bound on the number of VMs used by SHA(ε) for handling the total load $\tilde{\Lambda}_{i,j}^v + \bar{\Lambda}_{i,j}^v$.

First, note that the latency constraint of each top job contributing to $\tilde{\Lambda}_{i,j}^v$ is exactly $\frac{1}{\bar{\mu}-\lambda_{\min}(1+\varepsilon)^{j+1}}$. Using similar arguments as those used to prove Lemma 4, we can conclude that the maximum load on any VM running any such job is $\lambda_{\min}(1+\varepsilon)^{j+1}$, which we view as the *size* of the VMs used by SHA(ε) for handling these jobs. Recall that SHA(ε) can place jobs fractionally, and it does not place jobs with different latency range on the same VM. It follows that all VMs handling such jobs at layer ℓ, except for at most one, are completely full (recall that SHA(ε) is optimal and therefore, without loss of generality, can be assumed to place all jobs at a level in a single node at that level). Then, all these full VMs have a load equal to their size. Consequently, the number of full VMs required by SHA(ε) for handling jobs contributing to $\tilde{\Lambda}_{i,j}^v$ is at least: $\left\lfloor \frac{\tilde{\Lambda}_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^{j+1}} \right\rfloor \geq \frac{\tilde{\Lambda}_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^{j+1}} - 1$, which translates into

$$\tilde{\phi}(\text{SHA}_{i,j}^v) \geq \left(\frac{\tilde{\Lambda}_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^{j+1}} - 1\right)\kappa^\ell, \quad (9)$$

where $\tilde{\phi}(\text{SHA}_{i,j}^v)$ is the cost associated with running only the *full* VMs used by SHA(ε). Hence, we have:

$$\begin{aligned}\phi(\text{c-REShare}_{i,j}^v) &\leq 2(1+\varepsilon)\left(\frac{\tilde{\Lambda}_{i,j}^v}{\lambda_{\min}(1+\varepsilon)^{j+1}} - 1\right)\kappa^\ell \\ &\quad + [(2n+2)(1+\varepsilon)+1]\kappa^\ell \quad (10) \\ &\leq 2(1+\varepsilon)\tilde{\phi}(\text{SHA}_{i,j}^v) + [(2n+2)(1+\varepsilon)+1]\kappa^\ell,\end{aligned}$$

where the first inequality follows from (8), and the second from (9). By the relaxation property of SHA(ε), the overall cost of VMs used by SHA(ε), and working at full computation capacity, serves as a lower bound on the cost of OPT. Summing over all nodes i, VNFs v, and latency ranges j, we get:

$$\begin{aligned}&\phi(\text{c-REShare}(\varepsilon)) \leq \quad (11) \\ &\quad 2(1+\varepsilon)\phi(\text{OPT}) + ((2n+2)(1+\varepsilon)+1)J_\varepsilon\,|V|\sum_i \kappa^\ell.\end{aligned}$$

Since $J_\varepsilon \leq \log_{(1+\varepsilon)} \frac{\bar{\mu}}{\lambda_{\min}} - 1$, Eq. (11) implies that

$$\lim_{\phi(\text{OPT})\to\infty} \frac{\phi(\text{c-REShare}(\varepsilon))}{\phi(\text{OPT})} \leq 2(1+\varepsilon)\,,$$

thus completing the proof. □

V. Dynamically adjusting ε: REShare

In this section, we present the REShare algorithm, which dynamically adjusts ε in order to optimize the deployment of service instances as the service request load varies arbitrarily over time. The design of REShare draws upon the analysis of the competitive ratio of c-REShare(ε). Our analysis indicates that we get a better *multiplicative* competitive ratio when decreasing ε, implying that, ideally, for infinite duration requests, and ever increasing load, it is best to set ε as small as possible. Our analysis also shows that the *additive* terms in the competitive performance of c-REShare(ε) increase as ε decreases. Thus, using very small values of ε may end up being inefficient when the load is small. Consequently, REShare starts with a large ε to minimize the constant overheads, and, as the load increases, it reduces ε to optimize the asymptotic competitive ratio. Finally, since REShare explicitly deals with time-varying workloads, it increases ε when the total load declines, as the constant terms have a larger impact on its performance.

To do so, REShare simulates our shadow strategy SHA(ε) alongside the actual decisions it performs and uses the cost of SHA(ε) as an indicator of the system load, which is then leveraged to dynamically adjust the value of ε being used by REShare. Specifically, as the service request load increases, c-REShare(ε) gets asymptotically closer to SHA(ε) when ε is small, at the cost of a larger additive term due to the VMs partitioning into latency ranges L_j ($0 \leq j \leq J_\varepsilon$). This notion yields the main design criterion for REShare: ε

should be gradually reduced as the load grows, and instead increased when the load drops, as also depicted in Fig. 2. Importantly, we show that REShare has low complexity and it is asymptotically 2-competitive when requests have infinite duration, i.e., $\tau_r = \infty$; also, as shown in Sec. VI, it can effectively cope with very diverse, practical scenarios.

REShare can be seen as a way to apply different versions of c-REShare(ε), constantly adjusting the value of ε as the service load varies over time. Specifically, REShare begins by running c-REShare(ε) with a large initial value of ε; at the same time, it simulates SHA(ε) and keeps track of its cost. When the load goes beyond a threshold that, as detailed below, depends on the ratio between c-REShare(ε)'s and SHA(ε)'s costs, we keep track of the current load and reduce ε. This action is performed again and again as long as the load increases, following the arrival of new requests.

Once the load drops below the previous load mark, which is essentially due to requests leaving the system, we increase ε to its previous value. This approach is applied repeatedly as long as the load decreases.

Finally, we remark that, although we consider that service instances arrive and leave, their traffic load remains constant during their lifetime. REShare can also cope with time-varying values of λ_r. In particular, one can look at a change in the value of λ_r as if the current service instance left and another, exhibiting the new value of λ_r, arrived.

A. Algorithm description

To formally define REShare, we use the following notation. Let t_0 be the arrival time of the first request. We then look at sequence σ as the concatenation of subsequences $\sigma_1, \sigma_2, \ldots$, such that σ_q is the sequence of requests arriving/departing in interval $I_q = [t_{q-1}, t_q)$, for $q \geq 1$. Intervals are periods of time during which ε remains unchanged: for every I_q, c-REShare(ε) uses a given ε to determine the latency ranges, as described in Alg. 1. We indicate such value by $\varepsilon_{h(q)}$, where $h(q)$ denotes the index of the level of the system load at the beginning of I_q. The algorithm keeps track of the most recent load threshold, associated with index $h(q)$, throughout its execution, using parameter $T_{h(q)}$. Also, let us define:

$$Y_q = \tilde{\phi}(\text{SHA}(\varepsilon_{h(q)}), \sigma_q)\,, \tag{12}$$

$$Z = [(2n+2)(1+\varepsilon^*)+1] \log \frac{\bar{\mu}}{\lambda_{\min}} |V| \sum_i \kappa^\ell\,, \tag{13}$$

where, similarly to the proof of Theorem 1, $\tilde{\phi}(\text{SHA}(\varepsilon_{h(q)}), \sigma_q)$ is the cost associated with running only the full VMs used by SHA($\varepsilon_{h(q)}$). Here, $\varepsilon^* > 0$ is an initial value that satisfies $\varepsilon^* \geq \varepsilon_{h(q)}$, $\forall q$. We remark that the cost in (12) refers to the end of the interval I_q and that Z represents the second term in Eq. (10), which is independent of both q and $h(q)$. For every request r arriving or departing during I_q, we let σ_q^r denote the subsequence of request arrivals and departures in σ_q up to (and including) the arrival/departure of r. We further extend the above notation and define $Y_q^r = \tilde{\phi}(\text{SHA}(\varepsilon_{h(q)}), \sigma_q^r)$. At any time t, let $\tilde{Y}_p$ be the value of Y_q for the last interval in which $\varepsilon = \varepsilon_p$, over all intervals I_q ending no later than t. We then define:

$$C_q = \frac{Z}{\varepsilon_{h(q)} \log(1+\varepsilon_{h(q)})}\,, \tag{14}$$

$$S_q = \frac{1}{\varepsilon_{h(q)}} \sum_{p=1}^{h(q)-1} (2+3\varepsilon_{h(p)}) \tilde{Y}_p\,. \tag{15}$$

Note that during interval I_q, both C_q and S_q are fixed.

Algorithm 4 REShare

1: init $q = 1$; $h(1) = 1$; $\varepsilon_{h(1)} = \varepsilon^*$; $T_{h(1)} = 0$
2: **for each** service request r arriving or departing **do**
3: **handle** r in c-REShare($\varepsilon_{h(q)}$)
4: **update** the cost r in SHA($\varepsilon_{h(q)}$)
5: **if** $Y_q^r \geq \max\{C_q, S_q\}$ **then** ▷ arrival, cost (load) above threshold
6: $\varepsilon_{h(q)+1} \leftarrow \varepsilon_{h(q)}/2$ ▷ reduce ε
7: $T_{h(q)+1} \leftarrow \Lambda$ ▷ set load threshold
8: $\tilde{Y}_{h(q)} \leftarrow Y_q^r$
9: $h(q+1) \leftarrow h(q)+1$ ▷ update load level index
10: $q \leftarrow q+1$
11: **else if** $\Lambda < T_{h(q)}$ **then** ▷ departure, load below threshold
12: $h(q+1) \leftarrow h(q)-1$ ▷ update load level index
13: $q \leftarrow q+1$

REShare is formally defined in Alg. 4. REShare takes as input the value ε^*, which is the initial value of ε and is used to define Z. For each $q = 1, 2, \ldots$, the subsequence σ_q will be implicitly defined during the execution of REShare, according to the requests r handled by the algorithm between consecutive updates to the value of ε. Index $h(q)$, instead, keeps track of the evolution of the system load (by updating $T_{h(q)+1} = \Lambda$ in line 7, with Λ being the current system-wide load), and of the corresponding value of ε given as input to c-REShare, which is used as a subroutine within REShare (c-REShare($\varepsilon_{h(q)}$)). It follows that, by calling c-REShare with different values of the parameter $\varepsilon_{h(q)}$, REShare can adjust to the traffic load as this changes over time, always using the most appropriate values of ε.

Assume that the condition in Line 5 holds for some q and request r arriving or departing during σ_q. Since the right-hand side of the condition is fixed, this implies that Y_q^r has increased due to handling r, which means that r has arrived at the system, causing Y_q^r to increase beyond the value of the right-hand side. Since $Y_q^r \geq C_q$, along with the fact that in Line 8 $\tilde{Y}_{h(q)}$ is set to be Y_q^r, we have:

$$\varepsilon_{h(q)} \tilde{Y}_{h(q)} \geq Z \frac{1}{\log(1+\varepsilon_{h(q)})}. \tag{16}$$

Intuitively, using the insight derived from the analysis presented in Sec. IV, Eq. (16) implies that a mere $\varepsilon_{h(q)}$ fraction of the cost $\tilde{\phi}(\text{SHA}(\varepsilon_{h(q)}), \sigma_q)$ is already sufficiently larger than the cost incurred by REShare for handling requests in different

nodes than the ones used by SHA($\varepsilon_{h(q)}$), for input sequence σ_q. Also, since $Y_q^r \geq S_q$, it follows that:

$$\varepsilon_{h(q)}\tilde{Y}_{h(q)} \geq \sum_{p=1}^{h(q)-1}(2+3\varepsilon_p)\tilde{Y}_p. \quad (17)$$

Eq. (17) means that a fraction $\varepsilon_{h(q)}$ of $\tilde{Y}_{h(q)}$, i.e., a fraction $\varepsilon_{h(q)}$ of the cost $\tilde{\phi}(\text{SHA}(\varepsilon_{h(q)}), \sigma_q)$, is already sufficiently larger than the cumulative cost of SHA over the previous time intervals to warrant a change of ε. We note that these lower bounds on the *cost* of SHA($\varepsilon_{h(q)}$) are commensurate to the *load* encountered by the system (both by SHA($\varepsilon_{h(q)}$) and REShare), during interval I_q, and they come in handy for the analysis of REShare's competitive ratio.

When service requests expire and make the overall system load drop below the previous threshold (Line 11), the algorithm reverts to using the previous load level index $(h(q)-1)$, and, consequently, its corresponding value of ε.

B. Complexity and competitive ratio analysis

We first observe that the complexity of REShare is remarkably low: from inspection of Algs. 1–4 and considering the complexity of SHA(ε), the total complexity of REShare is $O(B|\mathcal{V}_r^s|)$ where B is the number of VMs currently used.

In the remainder of this subsection, we assume that requests have unlimited τ_r, i.e., there are no departures, and we analyze the competitive ratio of REShare in such a setting. In particular, under this assumption, we have that $h(q) = q$ for all q. For simplicity, we henceforth use index h to represent both q and $h(q)$. Furthermore, under these settings $\tilde{Y}_h = Y_h$, thus we simply use Y_h to denote either $\tilde{Y}_h$ or Y_h.

To prove REShare's competitive ratio, we proceed as follows. First, let us recall the observations made in Sec. V-A, i.e., whenever the condition in Line 5 of Alg. 4 holds, the overall load handled by the system at that point is significantly higher than the load at the beginning of the interval. In such a case, we reduce the value of ε to be used in the subsequent interval (Line 6). Since index h serves as a proxy to the overall load in the system, this update rule for the value of ε implies that $\lim_{h\to\infty}\sum_{p=1}^{h}\phi(\text{SHA}(\varepsilon_p,\sigma_p)) = \infty$ iff $\lim_{h\to\infty}\varepsilon_h = 0$. This is due to the fact that ever-decreasing values of ε yield ever-narrower latency ranges, hence a higher number of used VMs. Second, c-REShare(ε_h), as well as SHA(ε_h), handle the requests arriving in different intervals using a new series of latency ranges, hence they cannot reuse already deployed VNFs. Thus, the costs associated with different intervals I_h can be considered separately. Third, we prove the following lemma, which is the key result to derive the asymptotic competitive ratio of REShare.

Lemma 5. *For every $k = 1, 2, \ldots,$*

$$\sum_{h=1}^{k}\phi(\textit{REShare},\sigma_h) \leq (2+4\varepsilon_k)\sum_{h=1}^{k}\tilde{\phi}(\textit{SHA}(\varepsilon_h),\sigma_h).$$

Proof. By the definitions of Y_h and Z given in Eqs. (12)-(13), Eq. (10) implies (after changing the logarithm base) that

$$\phi(\text{REShare},\sigma_h) \leq (2+2\varepsilon_h)Y_h + Z\frac{1}{\log(1+\varepsilon_h)}. \quad (18)$$

By plugging Eq. (16) into Eq. (18), we obtain

$$\phi(\text{REShare},\sigma_h) \leq (2+3\varepsilon_h)Y_h. \quad (19)$$

By summing over h in (19) and using (17), we get

$$\begin{aligned}\sum_{h=1}^{k}\phi(\text{REShare},\sigma_h) &\leq (2+3\varepsilon_k)Y_k + \sum_{h=1}^{k-1}(2+3\varepsilon_h)Y_h \\ &\leq (2+4\varepsilon_k)Y_k. \quad (20)\end{aligned}$$

By the definition of Y_k, the thesis follows. □

The following theorem is an immediate corollary of Lemma 5, for the case of unbounded τ_r.

Theorem 2. *When τ_r is unbounded, REShare is asymptotically 2-competitive.*

Proof. By the update rule of ε_h, in line 6 of REShare, we have that $\lim_{h\to\infty}\varepsilon_h = 0$. By using Lemma 5, this therefore implies that $\lim_{h\to\infty}\frac{\phi(\text{REShare})}{\sum_{p=1}^{h}\tilde{\phi}(\text{SHA}(\varepsilon_p,\sigma_p))} \leq 2$. Since $\sum_{p=1}^{h}\tilde{\phi}(\text{SHA}(\varepsilon_p,\sigma_p))$ is a lower bound on $\phi(\text{OPT})$, the theorem follows. □

VI. Numerical results

In this section, we describe the scenarios we use for our performance evaluation (Sec. VI-A), the workloads we consider (Sec. VI-B), the benchmark solutions we compare against (Sec. VI-C), and the results we obtain (Sec. VI-D).

A. Reference scenarios

We consider two hierarchical topologies akin those originally proposed in [31] and used in many research works thereafter. The first is organized in three layers [32]: (i) *edge*, closest to the users (leaf nodes) but the most expensive (normalized, per-VM fixed cost of 7.5); (ii) *aggregation*, cheaper than edge (normalized, per-VM fixed cost of 2.5) but incurring a moderate extra traffic forwarding latency; (iii) *cloud*, cheapest (normalized, per-VM fixed cost of 1) but associated with the longest extra latency. The second is a four-layer topology, including a *fog* layer that incurs a per-VM fixed cost of 10. Cost figures are obtained from [26], presenting the technical and economic aspects of datacenters of different sizes, which can be placed at different network segments.

Each VM has a computing capability of $\bar{\mu} = 100$ packets/ms, and a per-packet proportional cost of 1/100th its fixed cost. This reflects findings reported in [25], i.e., that idle VMs consume roughly half as much power as fully-utilized ones (with the latter incurring both fixed and proportional costs). In the three-layer scenario, the extra latency associated with the aggregation and the cloud layer is, respectively, 15 ms and 30 ms. In the four-layer one, the fog layer has no extra latency, while the latency of all other layers is increased by 5 ms.

B. Workloads

We consider different workloads for our performance evaluation from three of the main application realms of slicing-enabled networks: connected vehicles, smart factories, and cloud-edge computing.

TABLE III
SERVICES TARGET DELAY AND VNF COMPLEXITY

VNF	θ_v	VNF	θ_v
Virtual comm. sub-slice		*CT – delay target 50 ms*	
eNB	1	Car information management (CIM)	10
EPC PGW	1	CT server	8
EPC SGW	1	CT database	1
EPC HSS	1	*EN – delay target 1 s*	
EPC MME	10	Video origin server	10
		Video CDN	3
ICA – delay target 20 ms		*Smart-factory – delay target 100 ms*	
Car information management (CIM)	7	Robot controller	10
Collision detector	10	Motion planning	10
Car manufacturer database	1	Configuration interface	5
Alarm generator	1	Digital twin application	10

Vehicular domain. We begin by considering the three-layer topology and the main services of the vehicular domain, presented in [33] and described in Tab. III:

- Intersection Collision Avoidance (ICA): vehicles periodically broadcast Cooperative Awareness Messages (CAMs) including their position, speed, and acceleration; a collision detector checks if any pair of vehicles are on a collision course and, if so, it issues an alert;
- Vehicular see-through (CT): cars display on their on-board screen the video captured by the preceding vehicle, e.g., a large truck obstructing the view;
- Entertainment (EN): passengers consume streaming content, provided with the assistance of a content delivery network (CDN) server.

In addition to their service-specific VMs, all services leverage a virtual communication sub-slice, as listed at the top of Tab. III. New service requests are generated whenever a new instance is needed, e.g., the service has to be deployed at a new location, or the crossing surveilled by ICA becomes more crowded, thus triggering a service scale out.

The computational requirements of the VNFs (i.e., their complexity) reported in Tab. III come from [33]; instead, the load we apply is synthetically generated, in order to demonstrate how our approach handles rapid demand fluctuations. Specifically, the service request process is as follows:

1) for the first 15 s of the time horizon under study, a new request arrives every second;
2) after that, and until 800 s from the beginning of the horizon, no further requests arrive;
3) between 800 and 1,000 s from the beginning, 1,000 more requests (5 per second) arrive;
4) those requests leave the system, at the same rate, between 1,000 and 1,200 s after the beginning of the horizon.

The first requests represent long-running services, which are active even during periods of low traffic. The subsequent request arrivals represent a sudden surge in vehicular activity, to which more service requests are associated, and an equally sudden decrease thereof.

Smart-factory domain. Digital twins are computer models of real objects, controlling the behavior of their physical counterparts. As detailed in [34], in smart-factory scenarios semi-autonomous robots are controlled by entities running within the network infrastructure. Also in this case we consider the three-layer scenario, along with the services specified in Tab. III. The main tasks to perform are: (i) fine-grained *control* of robot actions; (ii) *planning* of their actions and mobility; (iii) *configuration* of the robots; (iv) the *digital twin* itself. Comparing the smart factory to the vehicular service characteristics in Tab. III, one can notice how, while the end-to-end delays required in the two scenarios are comparable, the structure of the services is fairly different. Specifically, the smart-factory scenario only includes one service, including four VNFs, besides the virtual communication sub-slice. It follows that comparing the performance of REShare across these two use cases captures both quantitative differences and qualitative variability.

Real-world computing load. Finally, we move to the four-layer scenario and consider a real-world scenario where the demand comes from the GWA Materna trace [35], [36], which depicts the real-world evolution of the demand of a major cloud operator in Europe.

We also consider a different service request arrival process, where requests arrive and depart faster. Specifically:

1) for the first 15 s of the time horizon under study, a new request arrives every second;
2) after that, and until 780 s from the beginning of the horizon, no further requests arrive;
3) between 780 and 800 s from the beginning, 1,000 more requests (50 per second) arrive;
4) those requests leave the system, at the same rate, between 1,180 and 1,200 s after the beginning of the horizon.

C. Benchmark strategies

We compare REShare against two benchmarks. The first one is the shadow assignment used in Sec. IV-B. As discussed in Sec. IV-B, the cost of full VMs used by such a strategy is a lower bound on the optimum because it is allowed to place different parts of the same request at different hosts, which neither the optimum nor REShare are, of course, allowed to do, and it also handles relaxed delay constraints. Notice how SHA is not a concrete strategy that could be applied in a real-world scenario, but rather it serves as a proxy of the lower bound on the optimum cost. In other words, the closer to SHA a strategy is, the better that strategy performs.

The second benchmark, labeled *RelaxSoTA* in the plots, is an adaptation to our scenario of the highly influential works [3], [4]; importantly, the same approach has been later followed by further studies on edge computing [37], distributed machine learning [38], and energy-efficient mobile gaming [39]. Such approaches are based upon solving a convex relaxation of the placement problem every time a new request arrives, and making the placement decisions associated with the highest values of the relaxed variables. Notice that, unlike REShare, the RelaxSoTA approach may place VNFs of the same service at different layers.

In summary, we can say that both benchmarks have an advantage over REShare, respectively, the ability to split VNFs and to spread a service across multiple layers. Therefore, comparing REShare against such powerful alternatives yields additional relevance to our results.

Price-of-Dissimilarity (PoD). The PoD captures the additional computational capacity used because of the difference in delay constraints among the jobs served by the same VM (see also Sec. III). By the definition of the latency incurred on VM b running VNF v at speed μ_b, the delay incurred by *any* job assigned to b is $\frac{1}{\mu_b - \theta_v \Lambda(b)}$. We then define the PoD of VM b running on node i as: $\max_r \left\{ \frac{1}{D_r^v} - (\mu_b - \theta_v \Lambda(b)) \mid (r, v) \text{ is assigned to } b \right\}$. For any feasible assignment, the PoD of any VM b is non-negative; also, when *all* jobs assigned to b have the same delay constraint (as in SHA), it is possible to pick μ_b such that the PoD is zero.

D. Results

Vehicular domain. A fundamental aspect to investigate is the cost of REShare and its alternatives. In Fig. 4(left), we compare REShare, c-REShare(ε) with four different values of ε (identified by different markers), and the relaxation-based, state-of-the-art approach. For better readability, all values are normalized to the cost of SHA, which is why some cumulative costs in Fig. 4(right) appear to decrease. The gray, dotted line shows the number of requests in the system.

It can be immediately seen that REShare is substantially cheaper than the state-of-the-art benchmark we compare against and performs close to SHA, whose cost, we recall, is very close the optimum. Furthermore, REShare is cheaper than c-REShare(ε) for all ε values; this confirms the effectiveness of the strategy implemented in Alg. 4, whereby the value of ε is adjusted according to time-varying load conditions.

Fig. 4(right) presents the evolution of instantaneous (i.e., per time unit) costs during and around the load peak. Note that low values of ε are associated with a lower cost in high-traffic periods, while larger ε values yield lower costs in low-traffic periods, as highlighted in Fig. 2. This observation is consistent with how ε values determine how much PoD we tolerate on each open VM. When only a few requests are present, it is cheaper to tolerate a high PoD (large ε) since, otherwise, we would open an excessive number of (near empty) VMs. However, when the number of requests is high, the larger ε implies that we utilize the open VMs inefficiently. As for REShare, its cost is always close, albeit not equal, to the one of the *cheapest* instance of c-REShare(ε) due to the switching behavior of Alg. 4. Transitions between different ε values are marked in Fig. 4(right) by upwards- and downwards-pointing triangles.

We now characterize how ε affects the system's performance and cost. Fig. 5(left) shows how much of the services delay budget (D_r^s) is consumed by traffic forwarding (yellow areas) and processing (green areas). The traffic forwarding overheads are determined entirely by level choice to accommodate the job, which is the same for all ε values. However, larger values of ε result in shorter processing times. Shorter processing times, i.e., providing services *earlier* than their constraint, correspond to more computing resources unnecessarily provisioned and thus higher-than-needed costs.

This is confirmed by Fig. 5(center) and Fig. 5(right), highlighting how the VMs capacity is used. Green areas therein correspond to the load VMs have to serve, which cannot be reduced. The sum of orange and red areas correspond to the margin $\mu_b - \theta_v \Lambda(b)$ of the VMs (see Sec. IV-A); such a quantity must be larger than $\frac{1}{D_r^v}$ for all requests served by each VM. In particular, the orange areas correspond to the margin that VMs would have *if all requests they serve had the same latency constraints*, while the red ones correspond to the PoD. Finally, gray areas correspond to the difference $\bar{\mu} - \mu_b$ between the maximum and actually allocated VM capacities.

We can see that larger ε values are always associated with a higher PoD. If the load is low (as in Fig. 5(center)), larger ε values, implying more VM sharing and a higher PoD, may be an acceptable alternative to provisioning more VMs. This observation explains the behavior we observed in Fig. 4, where larger values of ε result in lower cost *in spite* of a higher PoD. For high load (Fig. 5(right)), limiting the PoD is instrumental in reducing the quantity of consumed resources. Specifically, from Fig. 5(right), we can see that the PoD is over 5% for RelaxSoTA, while it drops below 1% for REShare.

Smart-factory domain. Fig. 6 and Fig. 7 present the performance of REShare and its alternatives for the smart-factory application. Fig. 6 confirms that REShare yields the lowest *cumulative* cost (left plot), despite not necessarily being the cheapest solution at every point in time (right plot). It is also interesting to notice how the faster pace at which the load evolves also implies that c-REShare with low values of ε cannot catch up with REShare, and yield a substantially higher cumulative cost (first plot). REShare, on the other hand, can quickly go through all ε values and reach the optimal one, as shown by the green and red triangles at the bottom of the second plot of Fig. 6.

Fig. 7 shows how, despite the different services, smaller values of ε are consistently associated with a smaller PoD, though not necessarily with the lowest cost. By quickly reaching the right value of ε, REShare keeps the PoD below 1%, compared to 13% of RelaxSoTA.

Materna workload. The results for the real-world scenario based on the Materna trace are summarized in Fig. 8 and Fig. 9. We can observe a behavior that is effectively equivalent to Fig. 6 and Fig. 7, which further confirms how REShare works well with different loads and network topologies. As we can see from Fig. 9(right), the PoD for REShare is below 1%, compared to 21% of the state-of-the-art solutions. Throughout all scenarios, using REShare *in lieu* of state-of-the-art approaches consistently yields very significant cost savings, as summarized in Tab. IV. Interestingly, savings are higher in more complex scenarios, e.g., the real-world one.

TABLE IV
COST SAVINGS BROUGHT BY RESHARE W.R.T. THE STATE-OF-THE-ART (RELAXSOTA)

Scenario	Savings [%]
Vehicular/uniform	15
Smart factory	24
Materna (real-world)	26

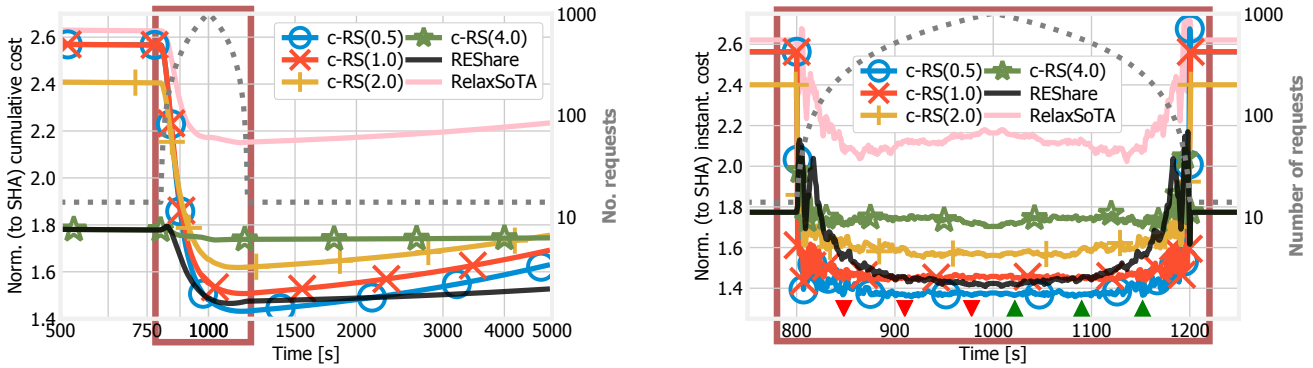

Fig. 4. Three-layer scenario, vehicular application. REShare and benchmark strategies: cumulative cost (left) and details of instantaneous cost during the load peak (right). In both plots, the dotted line corresponds to the load. In the right plot, upwards and downwards triangles at the bottom correspond to increasing and decreasing ε in REShare. Brown boxes denote the time period during which short-lived requests arrive and leave.

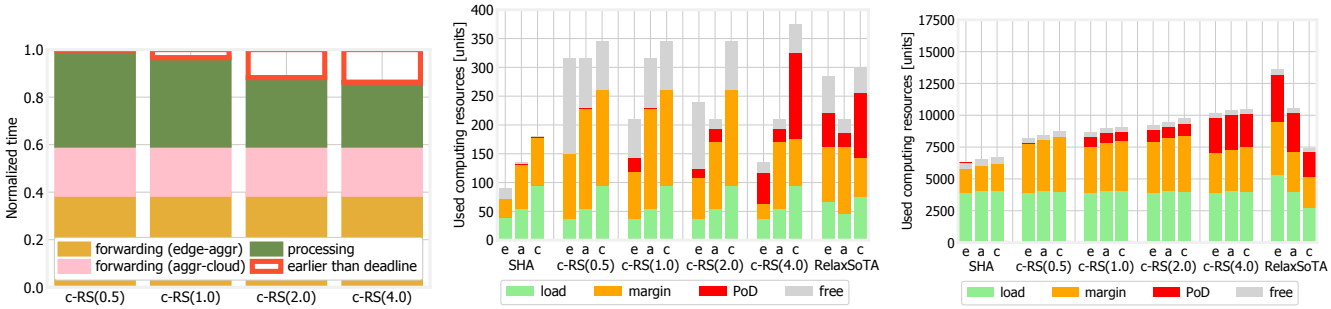

Fig. 5. Three-layer scenario, vehicular application. c-REShare(ε), with different values of ε (labeled by c-RS(ε) for short): actual service latency normalized to the target value (left); usage of computing resources for when traffic is low, namely, after the first 15 (long-running) requests arrive (center), and at peak load (right).

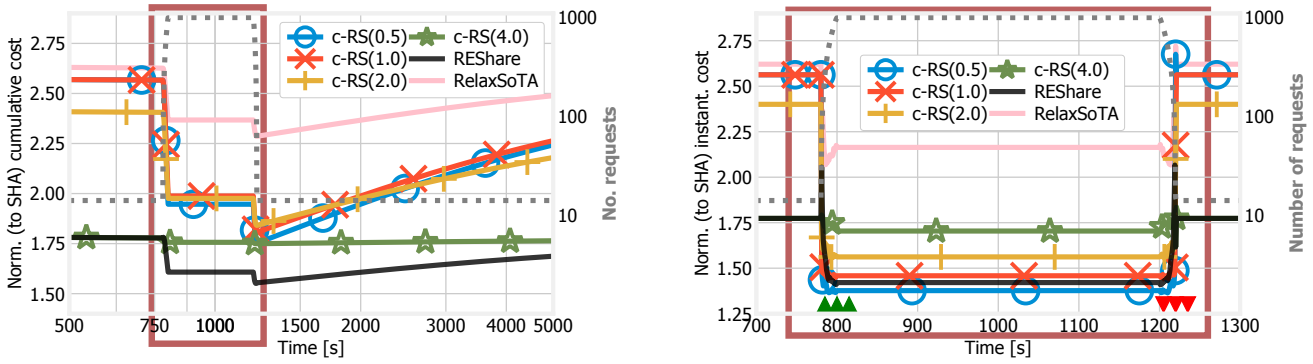

Fig. 6. Three-layer scenario, smart-factory application. REShare and benchmark strategies: cumulative cost (left) and details of instantaneous cost during the load peak (right). In both plots, the dotted line corresponds to the load. In the right plot, upwards and downwards triangles at the bottom correspond to increasing and decreasing ε in REShare. Brown boxes denote the time period during which short-lived requests arrive and leave.

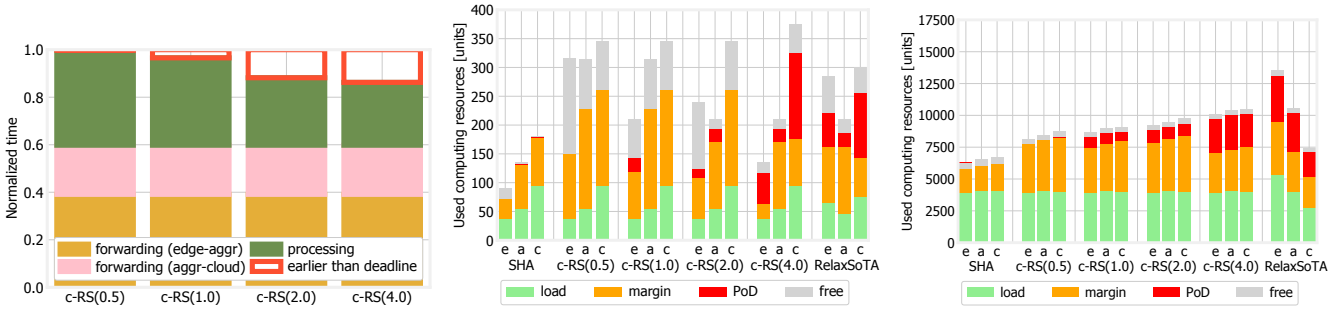

Fig. 7. Three-layer scenario, smart-factory application. c-REShare(ε), with different values of ε (labeled by c-RS(ε) for short): actual service latency normalized to the target value (left); usage of computing resources for the first 15 requests (center) and all requests (right).

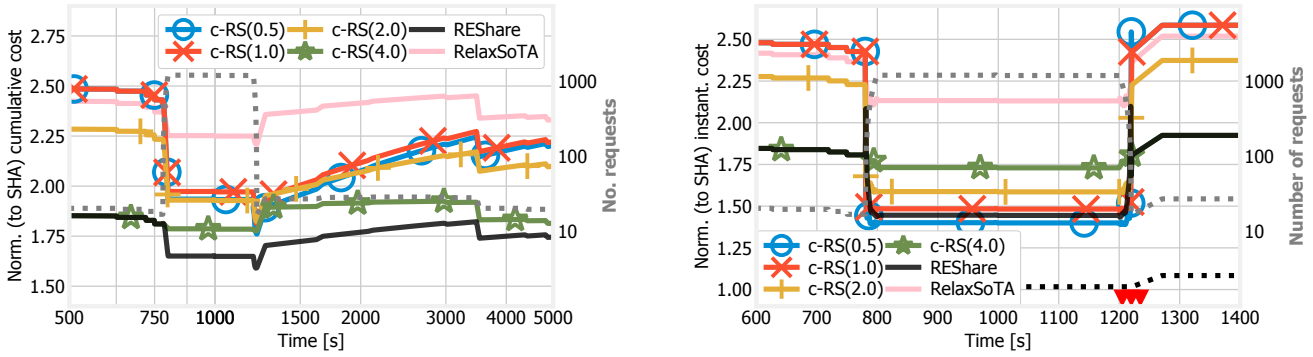

Fig. 8. Four-layer scenario, real-world workload based upon the Materna trace [35], [36]. REShare and benchmark strategies: cumulative cost (left) and details of instantaneous cost during the load peak (right). In both plots, the dotted line corresponds to the load. In the right plot, upwards and downwards triangles at the bottom correspond to increasing and decreasing ε in REShare.

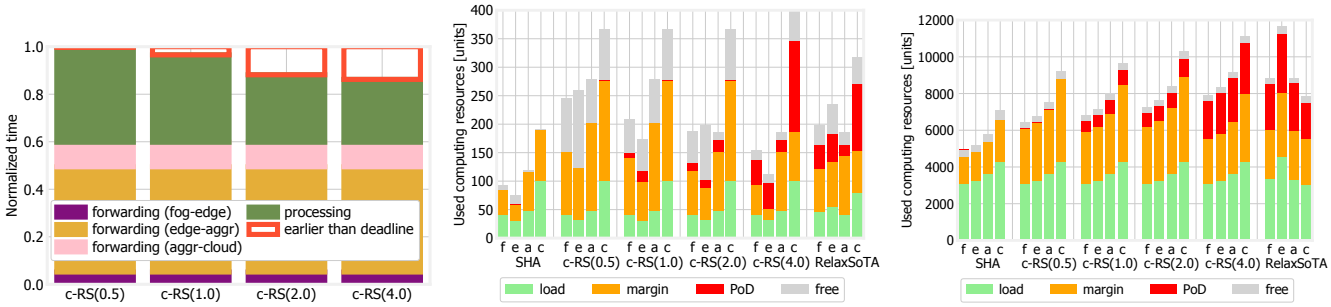

Fig. 9. Four-layer scenario, real-world workload based on the Materna trace [35], [36]. c-REShare(ε) performance for different values of ε (labeled by c-RS(ε) for short): actual service latency normalized to the target value (left); usage of computing resources for the first 15 requests (center), and all requests (right).

TABLE V
COMPARISON OF COMPETITIVE APPROACHES TO VNF PLACEMENT

Description	Model/approach	Latency	Dynamic	Guarantees	Refs
REShare	bin-packing, M/M/1	yes	yes	constant asymptotic competitive ratio: 2	our work
Capacitated NFV Location Algorithm	Generalized Assignment Problem (GAP)	no	no	bi-criteria: cost is at most 8 times the optimum, and constraints are violated by a factor of at most 8	[40]
SPR^3	multi-dimensional optimization; randomized approach	yes	no	competitive ratio (with high probability) of $4+\frac{3\log S}{R_n}$, where S and R_n are instance-dependent factors; constraints are satisfied in expectation	[4]
JASPER	multi-dimensional optimization; randomized approach	yes	no	competitive ratio (with high probability) of $3+\frac{2\log S}{\xi^\dagger}$, where S and $\xi^\dagger$ are instance-dependent factors; constraints are violated, with high probability, by at most a factor $4+\delta$, with $\delta \geq 0$ being a scenario-dependent quantity	[41]
GFT	MILP optimization	yes	no	asymptotic competitive ratio: $2+(1-o(1))\log m$, where m is instance-dependent	[42]
QNSD	multi-commodity-chain flow (MCCF)	yes	no	none for the full (integer) problem; $O(\varepsilon)$ competitive ratio for the fractional (relaxed) one	[9]
GSP-GRS	MILP optimization	yes	no	2 in special cases, none in general scenarios	[8]
GSP-SS	multi-scale scheduling	yes	yes	none	[43]
Online Throughput Maximization Algorithm	MILP optimization	yes	yes	$O(\log n)$, where n is instance-dependent	[44]

VII. Related Work

The pioneering work on VNF placement [40] casts the problem into a generalized assignment problem (GAP), and proposes a bi-criteria approximation thereto. Recent works [1]–[3], [5], [9], [45] widen the focus of the orchestration problem to include traffic routing as well as VNF placement. These studies present non-linear (and non-convex) problem formulations and, thus, resort to heuristics to solve the resulting problem. Other popular methodologies include graph theory [41], [46] and set-covering [42]. Several works also account for VNFs performing, the fact that VNFs can perform multiple tasks, e.g., caching [47], [48]. [8] follows a similar approach and jointly solves the problems of VNF placement and scheduling, i.e., which physical resources to use and when. In the same setting, [43] makes placement and scheduling decisions accounting for multiple resources, e.g., memory and storage, so as to reflect the requirements of the existing VNFs. Other works focus on specific services, e.g., [44] considers multicast streaming in MEC scenarios, and its peculiar requirements in terms of network latency and VNF capacity.

Most schemes work offline, i.e., all the service instances are known in advance. Among the few online approaches that deal with requests arriving at different times, [49], [50] incrementally update the current configuration, minimizing the changes to accommodate the new requests. In a similar setting, [3], [4] process service requests via a *randomized* approach. More recently, [51] performed placement offline and routing online.

While many works account for the fact that individual hosts (e.g., VMs) may have different capabilities and features, few consider layered topologies. Among those, [45], [52] focus on the choice between edge and cloud resources, and [53] studies the same problem with reference to caching, while [54] aims at jointly placing the VNFs and the data they need. Finally, several works characterize or predict service requests' arrival, thereby simplifying network management. Approaches include exploiting the traffic variability to reduce the amount of needed resources [55], using reinforcement learning to predict traffic [56], and estimating the resources needed by each request before admitting it [57]. Although REShare does not *require* any knowledge about the future evolution of the time demand, such information can be exploited, when available, to further improve its performance.

It is important to stress that, unlike REShare and c-REShare, existing works [3], [4], [6], [7] assume that VNF requirements are constant over time, and either are or are not satisfied by VMs. In Tab. V, we provide a summary of the comparison between previous work studying competitive approaches to VM placement, and our proposed solution. Importantly, ours is the *only* work featuring both a constant competitive ratio and the ability to deal with dynamic scenario, i.e., time-varying service demand.

A research problem closely related, albeit orthogonal, to REShare is represented by *predicting* the future demand for content and services. Examples include [16], where the authors envision using a deep neural network (DNN) to forecast the future demand, and make network orchestration decisions based upon such a forecast. In a similar spirit, [17] explores novel DNN architectures to better predict the load of cloud applications. It is important to stress that, whenever available, such predictions can seamlessly be integrated within REShare and further boost its performance.

VIII. Conclusions and Future Work

We addressed the problem of creating and scaling network slices while trying to reuse existing (sub-)slices across different services. We considered the availability of resources at different locations, including edge, aggregation, and cloud, and a time-varying system workload with service requests, arrivals, and departures. To effectively create and scale sub-slices, we proposed a low-complexity algorithm, which we proved to be asymptotic 2-competitive in the case of a non-decreasing load. Furthermore, numerical results obtained considering real-world services showed that our solution outperforms alternative approaches, for *time-varying workloads*, reducing the service cost by over 25%.

Future work will focus on extending our system model, most notably, by considering: (i) workload prediction as an approach to resource allocation for time-varying workloads, and (ii) VM migration, i.e., the possibility that VNFs move across different nodes during the service lifetime, and the consequent need to balance the activation of fewer VMs against migration cost. Furthermore, we will seek to implement REShare and assess its performance in real-world scenarios, first in small-scale testbeds and then through larger, cloud-based experiments. By so doing, we will be able to better prove the practical effectiveness of our solution.

References

[1] S. Agarwal, F. Malandrino, C. F. Chiasserini, and S. De, "VNF placement and resource allocation for the support of vertical services in 5G networks," *IEEE/ACM Trans. on Networking*, vol. 27, no. 1, pp. 433–446, 2019.

[2] J. Martín-Pérez, F. Malandrino, C. F. Chiasserini, M. Groshev, and C. J. Bernardos, "KPI guarantees in network slicing," *IEEE/ACM Trans. on Networking*, pp. 1–14, 2021.

[3] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Joint service placement and request routing in multi-cell mobile edge computing networks," in *IEEE INFOCOM*, 2019, pp. 10–18.

[4] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Service placement and request routing in mec networks with storage, computation, and communication constraints," *IEEE/ACM Trans. on Networking*, 2020.

[5] F. Malandrino, C. F. Chiasserini, G. Einziger, and G. Scalosub, "Reducing service deployment cost through VNF sharing," *IEEE/ACM Trans. on Networking*, vol. 27, no. 6, pp. 2363–2376, 2019.

[6] J. Liu, W. Lu, F. Zhou, P. Lu, and Z. Zhu, "On dynamic service function chain deployment and readjustment," *IEEE Trans. on Networks and Service Management*, vol. 14, no. 3, pp. 543–553, 2017.

[7] G. Moualla, T. Turletti, and D. Saucez, "Online robust placement of service chains for large data center topologies," *IEEE Access*, vol. 7, pp. 60 150–60 162, 2019.

[8] T. He, H. Khamfroush, S. Wang, T. La Porta, and S. Stein, "It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources," in *IEEE ICDCS*, 2018.

[9] H. Feng, J. Llorca, A. M. Tulino, D. Raz, and A. F. Molisch, "Approximation algorithms for the NFV service distribution problem," in *IEEE INFOCOM*, 2017.

[10] T. Lehman, X. Yang, N. Ghani, F. Gu, C. Guok, I. Monga, and B. Tierney, "Multilayer networks: an architecture framework," *IEEE Comm. Mag.*, vol. 49, no. 5, pp. 122–130, 2011.

[11] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration," *IEEE Comm. Surveys Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[12] "Building Telco Edge Infrastructure: MEC, Private LTE & vRAN," https://telco.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/microsites/telco/vmware-building-telco-edge-infrastructure.pdf, accessed Nov. 2021.

[13] M. A. Mollah, P. Faizian, M. S. Rahman, X. Yuan, S. Pakin, and M. Lang, "A comparative study of topology design approaches for HPC interconnects," in *IEEE/ACM CCGRID*, 2018, pp. 392–401.

[14] 5GROWTH Consortium, "Final design and evaluation of the innovations of the 5g end-to-end service platform," *5GROWTH deliverable 2.3*, 2022, https://5growth.eu.

[15] ETSI, "Open Source MANO (OSM)," accessed July 2020.

[16] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Deepcog: Cognitive network management in sliced 5g networks with deep learning," in *IEEE INFOCOM*, 2019.

[17] M. Xu, C. Song, H. Wu, S. S. Gill, K. Ye, and C. Xu, "esDNN: Deep neural network based multivariate workload prediction in cloud computing environments," *ACM Transactions on Internet Technology*, 2022.

[18] C.-H. Hong and B. Varghese, "Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–37, 2019.

[19] M. J. Neely, E. Modiano, and C.-P. Li, "Fairness and optimal stochastic control for heterogeneous networks," *IEEE/ACM Trans. on Networking*, vol. 16, no. 2, pp. 396–409, 2008.

[20] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center," in *IEEE CLOUD*, 2010, pp. 370–377.

[21] J. Prados, P. Ameigeiras, J. J. Ramos-Munoz, J. Navarro-Ortiz, P. Andres-Maldonado, and J. M. Lopez-Soler, "Performance modeling of softwarized network services based on queuing theory with experimental validation," *IEEE Trans. on Mobile Computing*, vol. 20, no. 4, pp. 1558–1573, 2021.

[22] J. Prados-Garzon, J. J. Ramos-Munoz, P. Ameigeiras, P. Andres-Maldonado, and J. M. Lopez-Soler, "Modeling and dimensioning of a virtualized mme for 5g mobile networks," *IEEE Trans. on Vehicular Technology*, vol. 66, no. 5, pp. 4383–4395, 2017.

[23] L. Kleinrock, *Theory, Volume 1, Queueing Systems*. USA: Wiley-Interscience, 1975.

[24] N. Kumar, G. S. Aujla, S. Garg, K. Kaur, R. Ranjan, and S. K. Garg, "Renewable energy-based multi-indexed job classification and container management scheme for sustainability of cloud data centers," *IEEE Trans. on Industrial Informatics*, vol. 15, no. 5, pp. 2947–2957, 2019.

[25] R. Morabito, "Power consumption of virtualization technologies: an empirical investigation," in *IEEE/ACM UCC*, 2015, pp. 522–527.

[26] 5G-TRANSFORMER Consortium, "Final system design and techno-economic analysis," *5G-TRANSFORMER deliverable 1.4*, 2019, http://5g-transformer.eu.

[27] "Timed MAFFT Alignment," *OpenBenchmarking*, 2020, https://openbenchmarking.org/test/pts/mafft.

[28] "FLAC Audio Encoding," *OpenBenchmarking*, 2019, https://openbenchmarking.org/test/pts/mafft.

[29] L. Epstein and A. Levin, "A robust APTAS for the classical bin packing problem," *Math. Program.*, vol. 119, no. 1, pp. 33–49, 2009.

[30] W. A. Trybulec, "Pigeon hole principle," *Journal of Formalized Mathematics*, vol. 2, 1990.

[31] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM*, 2016, pp. 1–9.

[32] J. Martín-Pérez, L. Cominardi, C. J. Bernardos, A. de la Oliva, and A. Azcorra, "Modeling mobile edge computing deployments for low latency multimedia services," *IEEE Trans. on Broadcasting*, vol. 65, no. 2, pp. 464–474, 2019.

[33] C. Casetti, C. F. Chiasserini, N. Molner, J. Martín-Pérez, T. Deiß, C.-T. Phan, F. Messaoudi, G. Landi, and J. B. Baranzano, "Arbitration among vertical services," in *IEEE PIMRC*, 2018, pp. 153–157.

[34] L. Girletti, M. Groshev, C. Guimarães, C. J. Bernardos, and A. de la Oliva, "An intelligent edge-based digital twin for robotics," in *IEEE Globecom Workshops*, 2020, pp. 1–6.

[35] A. Kohne, M. Spohr, L. Nagel, and O. Spinczyk, "FederatedCloudSim: a SLA-aware federated cloud simulation framework," in *ACM Workshop on CrossCloud Systems*, 2014, pp. 1–5.

[36] A. Kohne, D. Pasternak, L. Nagel, and O. Spinczyk, "Evaluation of SLA-based decision strategies for VM scheduling in cloud data centers," in *ACM Workshop on CrossCloud Infrastructures & Platforms*, 2016.

[37] S. Jošilo and G. Dán, "Joint wireless and edge computing resource management with dynamic network slice selection," *IEEE/ACM Transactions on Networking*, 2022.

[38] T. Q. Dinh, D. N. Nguyen, D. T. Hoang, T. V. Pham, and E. Dutkiewicz, "In-network computation for large-scale federated learning over wireless edge networks," *IEEE Transactions on Mobile Computing*, 2022.

[39] F. Spinelli, V. Mancuso *et al.*, "A migration path toward green edge gaming," in *IEEE WoWMoM*, 2022.

[40] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *IEEE INFOCOM*, 2015, pp. 1346–1354.

[41] S. Dräxler, H. Karl, and Z. Á. Mann, "JASPER: Joint optimization of scaling, placement, and routing of virtual network services," *IEEE Trans. on Networks and Service Management*, vol. 15, no. 3, pp. 946–960, 2018.

[42] Y. Sang, B. Ji, G. R. Gupta, X. Du, and L. Ye, "Provably efficient algorithms for joint placement and allocation of virtual network functions," in *IEEE INFOCOM 2017*, 2017, pp. 1–9.

[43] V. Farhadi, F. Mehmeti, T. He, T. F. La Porta, H. Khamfroush, S. Wang, K. S. Chan, and K. Poularakis, "Service placement and request scheduling for data-intensive applications in edge clouds," *IEEE/ACM Transactions on Networking*, 2021.

[44] Y. Ma, W. Liang, J. Wu, and Z. Xu, "Throughput maximization of nfv-enabled multicasting in mobile edge cloud networks," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[45] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Joint Service Placement and Request Routing in Multi-cell Mobile Edge Computing Networks," in *IEEE INFOCOM*, 2019.

[46] W. Ma, O. Sandoval, J. Beltran, D. Pan, and N. Pissinou, "Traffic aware placement of interdependent NFV middleboxes," in *IEEE INFOCOM*, 2017, pp. 1–9.

[47] J. Xu, L. Chen, and P. Zhou, "Joint service caching and task offloading for mobile edge computing in dense networks," in *IEEE INFOCOM*, 2018, pp. 207–215.

[48] M. Chen, Y. Hao, L. Hu, M. S. Hossain, and A. Ghoneim, "Edge-CoCaCo: toward joint optimization of computation, caching, and communication on edge cloud," *IEEE Wireless Comm.*, vol. 25, no. 3, pp. 21–27, 2018.

[49] T. Lukovszki, M. Rost, and S. Schmid, "It's a match!: Near-optimal and incremental middlebox deployment," *ACM SIGCOMM Comp. Comm. Rev.*, vol. 46, no. 1, pp. 30–36, 2016.

[50] ——, "Approximate and incremental network function placement," *Elsevier Journal of Parallel and Distributed Computing*, 2018.

[51] M. Blöcher, R. Khalili, L. Wang, and P. Eugster, "Letting off STEAM: Distributed runtime traffic scheduling for service function chaining," in *IEEE INFOCOM*, 2020.

[52] Y. Guo, A. L. Stolyar, and A. Walid, "Shadow-routing based dynamic algorithms for virtual machine placement in a network cloud," *IEEE Trans. on Cloud Computing*, vol. 6, no. 1, pp. 209–220, 2018.

[53] I. Cohen, G. Einziger, R. Friedman, and G. Scalosub, "Access strategies for network caching," in *IEEE INFOCOM*, 2019, pp. 28–36.

[54] K. Kamran, E. Yeh, and Q. Ma, "DECO: Joint computation, caching and forwarding in data-centric computing networks," in *ACM Mobihoc*, July 2019, p. 111–120.

[55] M. Bouet and V. Conan, "Mobile edge computing resources optimization: A geo-clustering approach," *IEEE Trans. on Networks and Service Management*, vol. 15, no. 2, pp. 787–796, 2018.

[56] V. Sciancalepore, F. Z. Yousaf, and X. Costa-Perez, "z-TORCH: An automated NFV orchestration and monitoring solution," *IEEE Trans. on Networks and Service Management*, vol. 15, no. 4, pp. 1292–1306, 2018.

[57] B. Han, V. Sciancalepore, D. Feng, X. Costa-Perez, and H. D. Schotten, "A utility-driven multi-queue admission control solution for network slicing," in *IEEE INFOCOM*, 2019, pp. 55–63.