Benefits of Pod dimensioning with best-effort resources in bare metal cloud native deployments

Benefits of Pod dimensioning with best-effort resources in bare metal cloud native deployments

Federico Tonini, *Member, IEEE,* Carlos Natalino, *Member, IEEE,* Dagnachew A. Temesgene, Zere Ghebretensaé, Lena Wosinska, *Senior Member, IEEE,* Paolo Monti, *Senior Member, IEEE*

***Abstract*—Container orchestration platforms automatically adjust resources to evolving traffic conditions. However, these scaling mechanisms are reactive and may lead to service degradation. Traditionally, resource dimensioning has been performed considering guaranteed (or request) resources. Recently, container orchestration platforms included the possibility of allocating idle (or limit) resources for a short time in a best-effort fashion. This paper analyzes the potential of using limit resources as a way to mitigate degradation while reducing the number of allocated request resources. Results show that a 25% CPU reduction can be achieved by relying on limit resources.**

***Index Terms*—Cloud native services, Pod dimensioning, Kubernetes, Best-effort resources, IaaS, Pod as a Service, soft-hard isolation, service degradation.**

I. Introduction

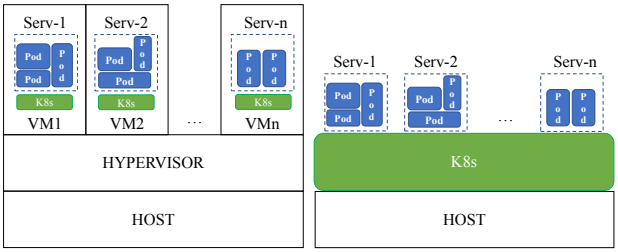
THE provisioning of services in today's communication paradigm generally involves two main entities, the *cloud provider* and the *service provider*. Cloud providers are responsible for the cloud infrastructure maintenance and updates, including managing physical (e.g., CPUs) and virtual resources (e.g., Virtual Machines (VMs), containers). These resources are usually rented out to service providers upon request. The cloud provider charges the service provider based on the amount and time resources are reserved/used. Service providers then use the rented VMs/containers to deploy end-user applications.

Cloud native technologies (e.g., containers) allow to easily develop, deploy, and manage services in the cloud [1]. Many networking applications such as monitoring, automation, Radio Access Network (RAN) and core virtualization have been shown to benefit from using cloud native technologies and containers [2]–[5]. Cloud native services are handled by cloud orchestrators like Kubernetes (K8s), a widely used open-source container orchestration platform [6]. In K8s, each service runs on a set of Pods. Each Pod is a collection of one or more containers, with a given amount of resources (e.g., memory, CPU, storage). The number of running Pods can be scaled (i.e., increased or decreased) over time to allow a service provider to match the time-varying end-user demands.

F. Tonini, C. Natalino, L. Wosinska, P. Monti are with the Electrical Engineering Department, Chalmers University of Technology, Gothenburg, Sweden. (e-mail: {tonini, carlos.natalino, wosinska, mpaolo}@chalmers.se).

D. A. Temesgene, Z. Ghebretensaé are with the Ericsson Research, Kista, Stockholm, Sweden. (e-mail: {dagnachew.azene.temesgene, zere.ghebretensae}@ericsson.com).

This work was supported by EUREKA cluster CELTIC-NEXT project AINET-ANIARA funded by VINNOVA.



(a) VM-based (b) Bare metal

Fig. 1. Deployment of cloud native services: VM-based vs. bare metal.

Pods can be deployed in VMs (Fig. 1a) in an *infrastructure-as-a-service* fashion, where each service provider needs to rent as many VMs as needed to compose its services. This approach ensures hard isolation of resources among different service providers, which need to pay for all the resources associated with the VMs, regardless of the number of running Pods. To fully take advantage of cloud native technologies, Pods of different services can be deployed directly over a common *bare metal* infrastructure without the need for a virtualization layer (Fig. 1b). This approach reduces the performance penalties introduced by hypervisors (e.g., for disk and network input/output operations) [7] while simplifying service deployment and operations [3]. Additionally, service providers can rent resources in a *Pod-as-a-service* fashion, paying only for what is needed to deploy and operate their Pods. Finally, soft resource isolation is also offered, allowing the use of both guaranteed (referred to as *request*) and shared (referred to as *limit*) resources. By doing so, idle (i.e., not used) resources initially set aside for one service can be used by Pods of another service when needed, in a best effort way.

Service providers decide the amount of resources assigned to each Pod they rent. If at any point in time, the resources/Pods are under-dimensioned and can not satisfy the end-user demands, the latter may experience degradation, e.g., an increased application response time, leading to a potential loss of revenue for the service provider [8]. On the other hand, if resources/Pods are heavily over-dimensioned, the service provider will pay for resources that are most of the time unused. Leveraging soft isolation might provide a third and interesting opportunity. A service provider can avoid overprovisioning by counting on the use of limit resources whenever needed. Since these resources are paid only when used, there is an evident advantage in terms of cost savings. On the other hand, since limit resources are not guaranteed, a service provider might face the possibility of higher degradation fees.

For this reason, it becomes crucial to investigate what is the potential cost vs. benefits of soft isolation.

Different techniques for the Pod dimensioning have been proposed in the past, considering mainly request resources and different scaling thresholds in VM-based deployments or bare metal deployments with only hard resource isolation [9]–[11]. The scaling can be based on machine learning and prediction techniques [12], [13], and include also application-related metrics (e.g., response time) [14]–[16] to improve the service performance. All these works focus on hard isolation and do not investigate the possibility of using soft resource isolation and limit resources to mitigate service degradation and reduce the overall costs. In this paper, we focus on the Pod dimensioning problem in bare-metal deployments, where the use of limit resources among Pods of different services is allowed. By leveraging on this, we analyze the potentials and limitations of using limit resources to reduce degradation without the need for over-dimensioning, by means of simulations. A cost analysis is performed by comparing this approach against a traditional scaling strategy relying only on request resources and shows when it is beneficial for a service provider to leverage limit resources.

II. Scenario Description and Use Case Example

In the following, we focus our attention on bare metal deployments of Pods handled by K8s (Fig. 1b). In K8s, resources such as CPU and memory are assigned by means of resource *request* and *limit* [17]. Request is the amount of guaranteed resources that each Pod can access at any time during its operation (hereinafter referred to as request resources). K8s assigns Pods to nodes based on the amount of request resources and the availability of resources on the node. A service provider pays for this type of resource even if they are not fully used by the running Pods (i.e., they are idle). On the contrary, limit is the amount of resources that are accessed on a first-come-first-served basis, in a best-effort fashion, only when two conditions are met: *(i)* a Pod needs more than the request resources, and *(ii)* there are unused resources at the node. We refer to this amount as limit resources. Limit resources usually take advantage of unassigned resources at a node, or unused request resources (idle) left free by other Pods. The amount of limit resources that Pods can access depends on the resource contention level at the node, which varies over time and depends on the amount of available resources, deployed Pods, and service requests. The service provider pays for limit resources only when accessed and for the time and quantity that has been used.

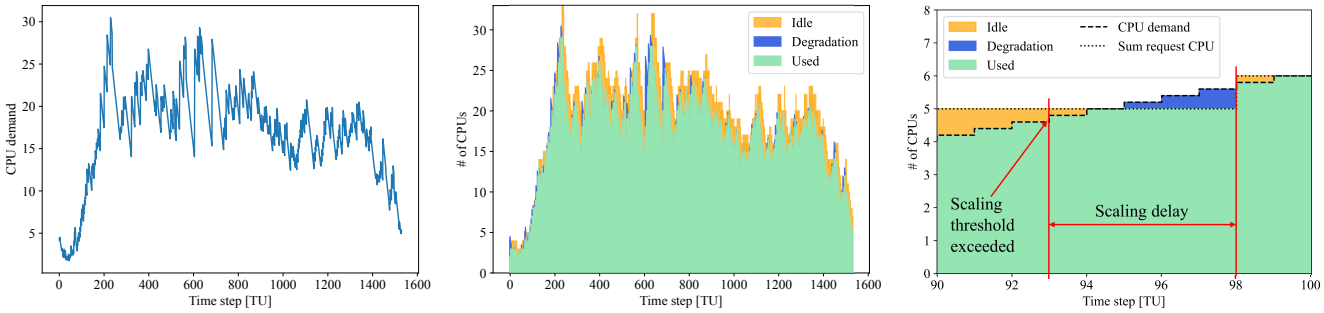
Service providers must solve the Pod dimensioning problem, i.e., to define the Pod's size (i.e., in terms of request and limit resources), the Pod's scaling parameters (e.g., desired average CPU usage), and the minimum and the maximum number of replicas. Pods can be replicated during the service operation to allow a service provider to match the time-varying needs of its end-users. Scale-out and scale-in operations rely on the monitoring capabilities of K8s and its built-in Horizontal Pod Autoscaler (HPA). The HPA uses periodical measurements related to a specific metric (e.g., CPU and/or RAM usage) from each one of the Pods [18] and computes the number of Pod replicas needed as follows:

$$dR = \left\lceil cR \cdot \frac{cMV}{dMV} \right\rceil, \tag{1}$$

where dR is the (new) desired number of replicas, cR is the current number of replicas, cMV is the current metric value, and dMV is the desired metric value (as specified by the service provider). In this work, we consider CPU as the resource and CPU usage as the metric to drive the scaling operations. In this case, cMV is the average CPU usage over all current Pods and dMV is the scaling threshold. In K8s, the threshold is indicated as the percentage of the request, and can be easily converted into the corresponding CPU amount dMV. To avoid frequent scaling operations, K8s establishes a default tolerance value by which the system does not scale if $0.9 < cMV/dMV < 1.1$ When scaling is triggered, some time is needed to adjust to the new desired state (i.e., to reach $cR = dR$). This time, referred to as *scaling delay*, is required to create/terminate Pods, update load-balancing components, and set up the service(s) within the Pod.

Ideally, a service provider would like to dimension and scale the number of running Pods in a way that allows renting just enough resources to match the CPU demand (i.e., the CPU required by the service provider to provide the services to the end users) over time while avoiding too many idle resources (i.e., CPUs paid for but unused) and degradation (i.e., number of CPUs that could not be allocated to Pods e.g., due to lack of resources). However, the scaling delay makes it difficult to always match the CPU demand, thus generating degradation for the users. As an example, let us consider the CPU demand over time shown in Fig. 2a. When the service is running, the number of Pods is adjusted by the HPA, and CPU allocation can be categorized as degradation, used, or idle. Fig. 2b shows the CPU allocation for a simple dimensioning case with 1 CPU request per Pod and without the possibility to use limit CPUs. The effect of the scaling delay (assumed to be 4 Time Units (TUs) in this case) can be observed when the CPU demand increases (Fig. 2c). At time step 93, 4.8 CPUs are used in total while 0.2 CPUs are idle. The threshold, set 4.25 CPUs, is exceeded, and $cMV/dMV > 1.1$. As a consequence, the desired number of replicas is updated using (1), triggering a scale out of 1 extra replica. Due to the scaling delay, this new replica is available at time step 98. The demand continues increasing and in the period between time steps 94 and 98 it exceeds the request CPUs, resulting in degradation.

Different countermeasures can be taken to mitigate degradation, depending on the level of degradation that each service can accept. During the dimensioning phase, the amount of request resources assigned to each Pod can be set to a higher value, allowing each Pod to access more resources. Another option is to select a low(er) scaling threshold, anticipating the scaling out process. However, both options lead to Pod over-dimensioning, potentially resulting in a higher amount of idle resources that must be anyway paid for. Another possibility is to allow the use of limit CPU resources. Considering the example in Fig.2, the area representing the degradation could be replaced, in part or in full, by limit CPUs. A service



(a) CPU demand over time (b) Detailed CPU allocation (c) Detailed scaling process

Fig. 2. Sample case of a cloud native service operated using K8s. CPU demand varies with end-user traffic. Time is discretized and expressed in Time Units.

provider could bet on their availability during operation to help reduce degradation, thus potentially lowering the amount of needed request CPUs.

III. Numerical Results

In order to evaluate the benefits of using limit resources while solving the Pod dimensioning problem, we developed a custom framework written in Python. The framework reproduces the HPA behavior explained in the previous section, with the following general assumptions. We assume a discrete amount of time instants in which we evaluate the service provider CPU demand, to be divided among the active Pods. We discretize the time in TUs that represents the monitoring cycles performed by K8s to obtain the metrics from the Pods and take actions (e.g., scaling). The CPU demand is considered to be an average over a time interval and evaluated at each cycle. This simplification is needed to avoid heavy simulation of run-time CPU resources scheduling, and it allows to measure average CPU degradation and idle resources in a similar fashion as K8s monitoring cycles.

A. Simulation settings

We consider a service provider with a CPU demand over time according to the workload pattern shown in Fig. 2a. This demand was gathered from Swedish University Network (SUNET) [19], converted into CPU load and augmented to mimic a real-world application. The traffic profile divides the 24 hours of a day into 1530 TUs. The obtained results are an average of 10 days. At each simulation, we add to each sample in Fig. 2a a random uniform value in the interval of $\pm 20\%$. We assume a service composed of a single Pod type, with two replicas deployed as the minimum number at time 0. At each time step, the CPU demand is split equally among the running replicas, simulating a perfect load balancing scheme. The number of replicas is also calculated at each time step according to (1), and the scaling delay is set to 4 TUs. Moreover, resources are measured in CPU$\times$TUs.

We use as a baseline a hard isolation deployment scheme over bare metal with the following configuration. The request CPU is set to 1 and no limit resources can be used, while the scaling threshold is set to 85% of the request CPU. We then analyze separately the two methods to mitigate degradation based on a higher threshold and a larger CPU request. The considered scaling thresholds are 85%, 75%, and 60%, while the Pod request values (hereinafter referred to as sizes) are $1, 2, 5$ CPUs.

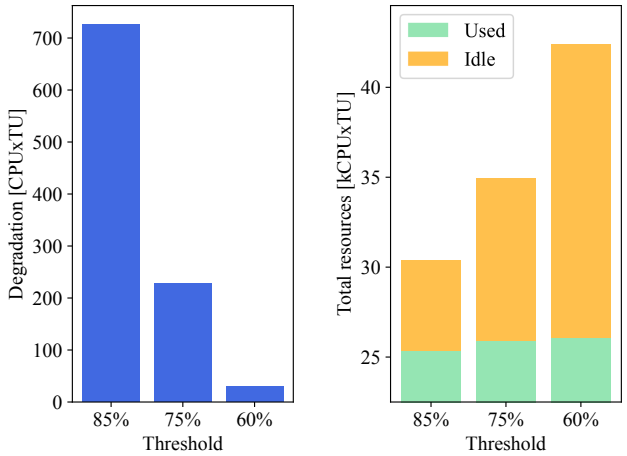
To assess the potential benefits of limit resources, we simulate the case in which Pods can use as many limit resources as necessary, as long as they are free. A parameter α is used to represent the amount of available limit resources as a portion of the request resources allocated for each Pod. For example, if $\alpha = 10\%$ and the CPU request is 1 CPU, each Pod can access up to 110% of the CPU request resources, i.e., 1.1 CPU.

B. Resource usage and degradation analysis

Fig. 3a shows degradation experienced by a service as a function of different scaling thresholds. We assumed that the Pod size is 1 CPU. We observe that lowering the threshold results in a lower degradation (i.e, from 726 to 30 [CPUxTU] when the threshold goes from 85% to 60%). This can be expected. With a more conservative value to trigger the scaling, Pods can access more CPU resources during the scale out process. However, there is a price to pay in terms of CPUs that stay idle (Fig. 3b). 12031 extra [CPUxTU] are required to reduce the degradation from 726 to 30 [CPUxTU], which is equivalent to 40% extra request resources required to reduce degradation by 96% with respect to the benchmark case. However, 94% of these extra resources are idle, as they are requested by the extra Pods, but not used most of the time.

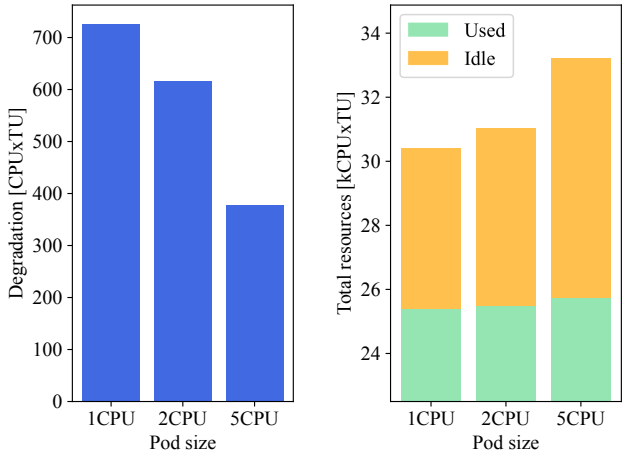
Fig. 4a reports the degradation for different (request) Pod sizes, with a fixed scaling threshold of 85%. Allocating Pods with more resources results in lower degradation. Fig. 4b reports the corresponding total resources. When the Pod size increases from 1 to 5 CPUs, the amount of total resources increases by 2822 [CPUxTU]. This corresponds to 9% extra request resources to reduce degradation by 52% with respect to the benchmark case.

In the following, we investigate whether limit resources can be used to mitigate degradation without impacting significantly idle resources. Fig. 5a depicts the degradation for different amount of limit resources that each Pod can access (α). The Pod request is 1 CPU and the scaling threshold 85%. When α=10%, the degradation decreases by 60% down to 294 [CPUxTU] with respect to the case with α=0%. When



(a) Degradation (b) Total resources

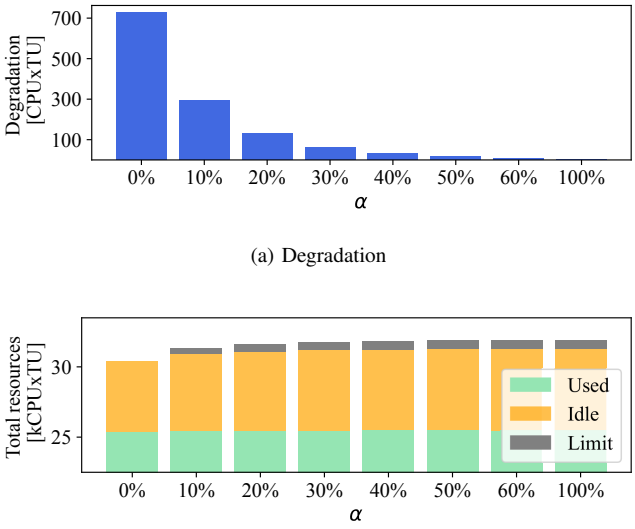
Fig. 3. Hard isolation case: degradation and total resources (in [CPUxTU]) for different scaling thresholds.



(a) Degradation (b) Total resources

Fig. 4. Hard isolation case: degradation and total resources (in [CPUxTU]) for different Pod sizes.

α=50%, the degradation is 17 [CPUxTU], a value similar to the case with a 60% threshold in Fig. 3a, and much lower than the values in Fig. 4a. Fig. 5b reports the total resources assigned to the service for different values of α. The total amount slightly increases with α, due to the larger amount of resources that can be accessed. When α=50% the total resources are 31858 [CPUxTU], adding only 1464 extra [CPUxTU] out of which 865 are request (used+idle) and 599 are limit. In this case, only 5% extra total resources are needed to reduce degradation by 98% with respect to the benchmark case. Compared to the value obtained with a threshold of 60% (Fig. 3b), 10566 [CPUxTU] resources can be saved, i.e., an improvement of 25%. These results show that using limit resources is potentially more resource efficient than relying on a high number of request resources. A service provider, instead of using a 60% threshold, could bet



(a) Degradation

(b) Total resources

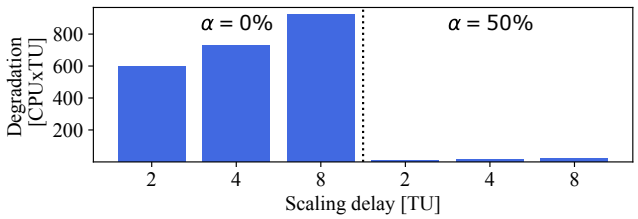
Fig. 5. Soft isolation case: degradation and total resources (in [CPUxTU]) for different amount of limit resources assigned to each Pod (α).

on having access to the required limit resources (α), limiting the extra resources assigned to Pods. However, relying only on limit resources does not give any guarantees, since the number of limit resources depends on the actual level of resource contention at the nodes where the Pods are running. Therefore, this approach is viable for service providers that want to improve service performance at a low cost, but can also accept some degradation. Conversely, service providers with strict constraints on degradation should rather rely on the resource over-provisioning strategies, such as using lower thresholds or larger Pod sizes. The actual level of resource contention is not under the control of the service provider as it depends on the effects of the runtime dynamics (e.g., how Pods are deployed, spikes in the CPU demand). The analysis of these aspects requires dedicated studies and is outside the scope of this paper.

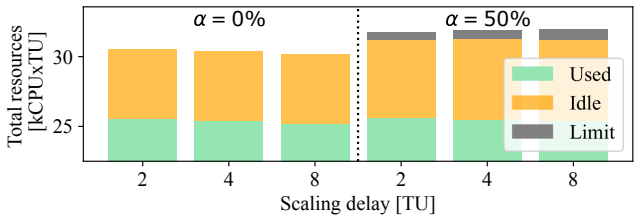
To analyze the effects of the scaling delay on the results, we simulated two additional cases, i.e., when the delay is 2 and 8 [TUs]. Figs. 6a and 6b report the degradation and total resources (in [CPUxTU]) for different scaling delays, when $\alpha = 0\%$ and $\alpha = 50\%$, the Pod size is 1 CPU and the scaling threshold is 85%. Results show that a lower scaling delay corresponds to a lower degradation. This is due to a faster response to CPU demand variations, increasing total resources. By using soft isolation, degradation is compensated with a small number of limit resources regardless of the scaling delay value, which confirms the effectiveness of this approach.

C. Cost analysis

To analyze the costs, we compare an approach based on limit resources (lim) with one that uses only request resources (req) and one that just accepts degradation (deg). The lim approach is beneficial if its cost (C_{lim}) is lower than or equal



(a) Degradation



(b) Total resources

Fig. 6. Degradation and total resources (in [CPUxTU]) for different scaling delays in [TUs], when $\alpha = 0\%$ and 50%.

TABLE I
AMOUNT OF REQUEST (R), LIMIT (L), AND DEGRADATION (D) IN [CPUXTU] FOR THE THREE CONSIDERED CASES (lim, deg, req).

	lim	deg	req
R	31858	30394	42425
L	599	0	0
D	17	726	30

to the deg cost (C_{deg}) and the cost of the req solution (C_{req}). The following system holds:

$$\begin{cases} C_{lim} \leq C_{deg}, & (2) \\ C_{lim} \leq C_{req}, & (3) \end{cases}$$

Each cost depends on the amount of request resources (R), limit resources (L), and degradation (D) and their unitary price (p_R, p_L, and p_D, respectively). For the lim case we have $C_{lim} = R_{lim}p_R + L_{lim}p_L + D_{lim}p_D$. Similar formulations can be derived for C_{deg} and C_{req}. We can use (2) and (3) to determine the values of p_R, p_L, p_D for which the use of the lim approach is beneficial. As an illustrative example, let us consider the case with α=50% in Fig.5 as lim, the benchmark as deg and the case with 60% threshold as req (first and last bars in Fig.3, respectively). The related amount of resources is reported in Tab.I. Let us consider the worst case for lim, i.e., the equalities in (2) and (3). By solving (2) for p_D and substituting it in (3) we find that $p_L = 18.02p_R$. Lower values of p_L make the lim approach appealing for a service provider compared to the req approach. Similarly, we can solve (3) for p_R and substitute it in (2) to find that $p_L = 1.04p_D$. For lower values of p_L, using lim is more beneficial than deg (i.e., compensating degradation with limit resources is less costly than accepting it).

IV. CONCLUSION AND FUTURE WORK

This work presents a performance analysis of different Pod dimensioning strategies in cloud native scenarios from a service provider perspective. A simulator has been developed to mimic K8s behavior and evaluate the performance of these strategies in terms of degradation and idle CPUs. Results show that degradation can be mitigated by using limit resources under the assumption that additional α% limit resources can be accessed by the Pods. In particular, a strategy based only on limit resources with α=50% can achieve the same level of degradation as a conventional strategy with a scaling threshold fixed to 60%, while requiring 25% less reserved CPUs. Moreover, savings can be achieved if the unitary price of a [CPUxTU] of limit resource is lower than 18 times the price of a request [CPUxTU] resource. Since limit resources are not reserved, this approach is viable for service providers that can tolerate some degradation in case the limit resources are not available. Nevertheless, intelligent Pod scaling strategies can be developed to compensate for this, e.g., by monitoring degradation and adding request resources only when needed. Studies from a cloud provider perspective on how to price resources and how to increase resource sharing are left as future work.

REFERENCES

[1] "CNCF," https://www.cncf.io/.
[2] RedHat, "Kubernetes on Bare Metal: The future of RAN," Tech. Rep., 2020.
[3] Ericsson AB, "Building a cloud native infrastructure," Tech. Rep., October 2020.
[4] L. Sanabria-Russo and C. Verikoukis, "A Cloud-Native Monitoring System Enabling Scalable and Distributed Management of 5G Network Slices," in *IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, 2021, pp. 42–46.
[5] S. Roy *et al.*, "A Cloud Native SLA-Driven Stochastic Federated Learning Policy for 6G Zero-Touch Network Slicing," in *IEEE International Conference on Communications (ICC)*, 2022, pp. 4269–4274.
[6] "Kubernetes," https://kubernetes.io/.
[7] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *IEEE International Conference on Cloud Engineering*, 2015, pp. 386–393.
[8] Akamai, "The state of online retail performance," Tech. Rep., 2017.
[9] E. Casalicchio, "A study on performance measures for auto-scaling CPU-intensive containerized applications," *Cluster Computing*, vol. 22, no. 3, pp. 995–1006, Sep 2019.
[10] V. Millnert and J. Eker, "Holoscale: horizontal and vertical scaling of cloud resources," in *IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 196–205.
[11] D. Balla *et al.*, "Adaptive scaling of Kubernetes Pods," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2020.
[12] F. Rossi *et al.*, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *IEEE International Conference on Cloud Computing (CLOUD)*, 2019, pp. 329–338.
[13] L. Toka *et al.*, "Adaptive AI-based auto-scaling for Kubernetes," in *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 599–608.
[14] C.-C. Chang *et al.*, "A Kubernetes-based monitoring platform for dynamic cloud resource provisioning," in *IEEE Global Communications Conference (GLOBECOM)*, 2017.
[15] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with SLA objective using Q-Learning," in *International Conference on Future Internet of Things and Cloud (FiCloud)*, 2018, pp. 85–92.
[16] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *IEEE International Conference on Web Services (ICWS)*, 2019, pp. 68–75.
[17] "Kubernetes Resources," https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.
[18] "Kubernetes Horizontal Pod Autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[19] "SUNET database," http://stats.sunet.se/, accessed: 2021-10-27.