Providing Realtime Support for Containerized Edge Services

WENZHAO ZHANG, College of Computer Science, Zhejiang University, China
YI GAO AND WEI DONG, College of Computer Science, Zhejiang University; Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China

Containers have emerged as a popular technology for edge computing platforms. Although there are varieties of container orchestration frameworks, e.g., Kubernetes to provide high-reliable services for cloud infrastructure, providing real-time support at the containerized edge systems (CESs) remains a challenge. In this paper, we propose EdgeMan, a holistic edge service management framework for CESs, which consists of (1) a model-assisted event-driven lightweight online scheduling algorithm to provide request-level execution plans; (2) a bottleneck-metric-aware progressive resource allocation mechanism to improve resource efficiency. We then build a testbed that installed three containerized services with different latency sensitivities for concrete evaluation. Besides, we adopt real-world data traces from Alibaba and Twitter for large-scale emulations. Extensive experiments demonstrate that the deadline miss ratio of time-sensitive services run with EdgeMan is reduced by 85.9% on average compared with that of existing methods in both industry and academia.

CCS Concepts: • **Networks → Network services**; • **Computer systems organization → Real-time system architecture**.

Additional Key Words and Phrases: Realtime Support; Containerized Edge Service; Cloud-Native

1 INTRODUCTION

The container is becoming popular for its productivity (e.g., accelerate development, improve consistency across environments), resilience (i.e., prevent single point failure), portability (i.e., move across nodes), scalability (e.g., scale up/down in terms of replica), etc. These benefits are widely recognized by cloud providers [8, 31, 49]. According to Gartner, by 2025, more than 85% of global organizations will be running containerized applications in production [25].

Edge systems usually hold multiple services for both cloud and end devices. Containerized Edge Systems (CESs) emerge to provide service isolation and facilitate seamless cloud-edge coordination. Recent edge systems such as EdgeX [21] and Azure IoT Edge [9] leverage the container as the backbone infrastructure.

Real-time computing is crucial for many CESs use cases:

- Automated monitoring and control systems at the edge [41]: Systems tasked with monitoring and controlling infrastructure, equipment, vehicles, and other physical assets at the edge must detect events and adjust in real time. Delayed detection, response, or control in these environments could have disastrous safety, operational or economic consequences before the issue or event was remediated from the cloud or a remote

This work is supported by the National Key R&D Program of China under Grant No. 2019YFB1600700, the National Natural Science Foundation of China under grant No. 62072396 and 62272407, "Pioneer" and "Leading Goose" R&D Program of Zhejiang Province under grant No. 2023C01033, and the National & Zhejiang Provincial Youth Talent Support Program. (Corresponding authors: Yi Gao and Wei Dong).
Authors' addresses: Wenzhao Zhang, College of Computer Science, Zhejiang University, 38 Zheda Rd, Hangzhou, China, wz.zhang@zju.edu.cn; Yi Gao and Wei Dong, College of Computer Science, Zhejiang University; Alibaba-Zhejiang University Joint Institute of Frontier Technologies, 38 Zheda Rd, Hangzhou, China, {gaoy,dongw}@zju.edu.cn.

1533-5399/2023/9-ART $15.00
https://doi.org/10.1145/3617123

Table 1. Service specifications under K8s settings, with the HPA and VPA disabled. The one-shot execution time is the stable average execution time for a specific service.

Specifications/Service		**LPR**	**VC**	**VD**
Quota Range	*CPU (Core)*	[0.8, 1]	[6.4, 8]	[0.64, 0.8]
	Mem (GB)	[0.16, 0.2]	[0.95, 1.2]	[0.14, 0.18]
Replica Limit		10	5	10
One-shot Execution Time (s)		0.75	9.5	0.03
Deadline (s)		0.80	10	0.04

center. Real-time operation using edge CESs is essential for such monitoring and control applications where rapid local response is imperative.

- Anomaly detection at the edge [47]: Real-time anomaly detection using CESs is needed to promptly detect and mitigate threats like equipment tampering, component malfunctions, network intrusions, or other undesirable events in remote environments. Delayed edge detection of such anomalies defeats the purpose and could allow harm to occur before the issue is observed or addressed.
- AR/VR/XR [39]: AR enabled by CESs could leverage real-time location data, camera inputs and computer vision to overlay interactive information and objects on the surrounding physical world. Rapid processing is needed to dynamically respond to a user's field of view, location and actions.

In above example cases, failure to meet real-time service constraints is not merely inconvenient but could profoundly compromise systems at the edge by delaying or preventing rapid autonomous response to urgent needs.

Provisioning proper amount of resources is the essence of providing real-time support in CESs as the ultimate goal of existing methods (e.g., priority-based scheduling, resource reservation, determinism, limited preemption, and interference avoidance [10, 13, 36]) is to make sure that the service can have enough resources (either CPU cores or others) to finish its request on time. Container engines like Docker provide fine-grained resource allocation capability via Cgroups [60].

Unfortunately, the ability to allocate resources only is not enough. Because the resource allocation plan is fixed for a service, which means that when the computing context of this service is changed (e.g., there is a workload surge or another service with higher-priority is started), the performance degrades. To deal with this degradation, system operators will have to re-allocate resource manually according to the highly change contexts. To avoid the tedious manual configuration, container orchestration frameworks like Kubernetes (K8s) emerge. K8s can monitor different metrics (CPU, memory, etc.) of a service and scale it up/down according to a predefined threshold set by the user, e.g., "adding two replicas if CPU utilization surpasses 80%". Although K8s can provide service QoS through container-level dynamic resource allocation, it is still hard to provide request-level real-time support, especially for services with short tasks (e.g., a few minutes).

An example scenario. Consider a bridge monitoring scenario. At the edge, we deploy three containerized edge services: License Plate Recognition (LPR), Vehicle Counting (VC), and Vibration Detection (VD). LPR is executed when there is an important event, e.g., hunting a fugitive's car. VC runs continuously to provide statistical information about the vehicle traffic on the bridge. VD keeps monitoring if there is an abnormal vibration happens. They are typical services running in edge nodes to give timely response. The LPR and the VC service are deployed on edge servers that are closer to the camera; the VD service is deployed on industrial personal computers at the edge that are closer to the vibration sensors (e.g., accelerometers). Table. 1 shows the specifications of these services together with the arbitrary deadline settings. We can see that they have distinct features, e.g., latency sensitivity and resource requirements. Specifically, LPR has both a medium execution latency (0.75s) and resource requirements (1 CPU, 200 MB memory), VC needs both a large execution latency

(9.5s) and resource requirements (8 CPU, 1200 MB memory), VD has a small execution latency (0.03s) and small resource requirements (0.8 CPU, 180 MB memory). According to the survey [65] conducted by Varghese et al., which covers 14 edge computing related benchmarks and includes 28 edge services, 85.71% of which are data analysis service with deep-learning or machine-learning models. The three services are also fall in this category. In summary, we believe the three services are good representative of services running in edge.

The timeliness of a service can be interpreted as the number of requests that do not miss the deadline of their belonging services in the long run. Our goal is to provide real-time support for these services. Unfortunately, existing methods can hardly achieve this goal (detailed discussions see §3.2 and §8).

Issues and root causes. *Issue 1: Request Unawareness*. Requests might constantly violate the deadline while the metric monitor can barely find out. The cause is two-fold: (1) *Requests can easily pile up* as many edge services are single-threaded or have a fixed-size queue. According to the survey conducted by Varghese et al. [65], 82.14% of the surveyed edge services are single-threaded [17, 29, 69] or queued [58]. The coming requests can be executed with assigned resource utilization but will suffer from high waiting latency. Nevertheless, K8s only knows that the resource utilization is under the threshold without worrying about the high deadline miss ratio of requests. For example, VC is a CNN-based single-threaded service that takes video stream as input and outputs the vehicle number. When 3 requests come together, the execution time for each of them is 9.5s, 19s (wait for 9.5s), and 28.5s (wait for 19s) respectively while the resource consumption remains the same. As a result, the overall deadline miss ratio of VC for these requests is 66.67%. (2) The metric sampling and autoscaling period are too long in comparison to request processing. K8s is designed for large-scale distributed cloud systems that host many long-lived services, whose primary concern is stability. K8s is not optimized for short-lived requests, which requires shorter monitoring periods. A short sync period will lead to unreliable resource metric value and fluctuating replica numbers, which is usually unacceptable. Consequently, requests might constantly miss the deadline due to the unmatched time scale. For example, the default scheduling and sampling period of K8s is 15s. If we send continuous tasks, say 1 request per 0.01 second, to VD, all 15 requests cannot meet their deadline before K8s can do anything.

Issue 2: Resource Wastage. K8s autoscaler can horizontally add/remove replica (i.e., horizontal pod autoscaler, HPA) or vertically increase/decrease the resource quota (i.e., vertical pod autoscaler, VPA) of a service. Unfortunately, the former will consume/release resources in the unit of pods[1] and the latter recommends resources only base on hourly-averaged utilization history, which usually lacks accuracy. For example, say K8s HPA decides to horizontally scale up LPR (add a replica) when it detects the CPU utilization surpasses 80%. Then resultant resource consumption will scale from 1 CPU, 0.2 GB memory to 2 CPU, 0.4 GB memory. However, the actual need to handle more requests of LPR may be 1.5 CPU and 0.25 GB memory. The K8s VPA tries to address this issue by increasing the CPU and memory quota individually rather than monotonically. Unfortunately, the stale information it leverages cannot afford to a highly changing environment.

Challenges. Tackling the issues above is not straightforward. *The challenge for tackling issue 1: Request modeling can hardly adapt to a highly dynamic environment.* It seems that system operators can simply implement a new plug-in [1] to detect request arrival for each service and achieve ms-level metric scraping via metric customization. Specifically, developers can write highly efficient Cgroups metric scraper and integrate it to CESs using K8s APIs [2, 3] or metric system such as Prometheus [4]. Although system operators can manage to make CESs become request aware, achieving real-time is still challenging because of the high request rate. Ideally, CESs should be able to provide real-time support for each service request, which requires highly lightweight but effective mechanisms. For example, CESs need to know current computing context (e.g., resource utilization), the coming workload (e.g., request concurrency), the estimated latency of processing these requests given different resource allocation plans, etc. Achieving this is non-trivial because of the requirement of thorough, foreseeable information about

[1]The pod is the smallest scheduling unit in K8s that contains one or more containers.

the service workload, resource requirement, and the corresponding latency. However, traditional model-based works [14, 43, 80] require tons of offline profiling and thus can hardly be adaptable to highly dynamic context changes (e.g., hardware, application types, and computational intensity).

The challenge for tackling issue 2: The search space of finding suitable resource quota for containerized services is exponentially large. Developers can also follow their intuition or make thorough profiling to configure the computing resources (e.g., CPU, memory) for each service. This straightforward method can be far from optimal as the developer can hardly know the relationship between the configured resources and the corresponding performance [71], which usually leads to over- or under-provisioning [78]. Existing schedulers for containers [6, 23, 28, 30, 34, 68, 74, 76, 78, 79] allocate resources in coarse-grained (e.g., replica number, CPU cores), which causes achieve high resource wastage. For example, say the ideal CPU requirement of the VD service is 0.8, existing methods tend to provision 1 CPU for it, leading to 0.2 CPU core wastage. What is worse, finding the optimal resource quota for each service is exhaustive if not impractical. If we take a step of 0.1, 50 MB in CPU and memory quota (with the maximum value of 4, 2000 MB, respectively), 1 for replica (with the maximum value of 10), the total number of possible policies for three services will be: $(40 * 40 * 10)^3 \approx 4.1 \cdot 10^{12}$.

Our approach. This paper proposes the EdgeMan system with novel techniques to solve the above issues in CESs by using unique characteristics. For *issue 1*, we propose a model-assisted lightweight scheduling algorithm to achieve request-aware decisions. Concretely, we use CES-specific features to build workload and latency models, providing accurate near-future information. Given that, we develop a priority-based scheduler to generate fast but effective decisions. For *issue 2*, we propose a progressive bottleneck-aware allocation mechanism to make time- and resource-efficient autoscaling policies. Specifically, we build a gradient-based model to describe the impact of a resource type. With the above models, we leverage a heuristic algorithm to efficiently search exponentially large decision space and give reasonable resource allocation policies.

We implement EdgeMan and evaluate its performance extensively. Results show that: (1) EdgeMan can reduce 85.9% deadline miss on average. (2) The resource utilization of EdgeMan is 81.8% higher than that of exiting works. (3) The policy searching efficiency of EdgeMan is 99.6% faster than that of baselines.

We summarize the contributions as follows:

- We propose EdgeMan, a real-time support solution for containerized edge services with both long- and short-lived tasks. We carefully formulate the problem, which is a mix-integer non-linear programming problem.
- We develop a request-level event-driven scheduling algorithm and a progressive bottleneck-metric-aware resource allocation mechanism to solve the request unawareness and poor resource utilization issue.
- We implement the prototype of EdgeMan together with a testbed that installs three latency-sensitive services. We evaluate EdgeMan through both extensive emulation with two real-world data traces from *Twitter* [56] and *Alibaba cluster* [7] as well as testbed execution.

2 SYSTEM MODEL AND PROBLEM FORMULATION

2.1 System Model

We first introduce the following notations.

- V, E, S. We use them to denote the set of nodes, communication links, and edge services in the network.
- $i, j, k, v_i, e_{i,j}, s_k$. We use v_i to denote a node; $e_{i,j}$ indicates if there is a communication link between v_i and v_j; s_k denotes a service.
- $b_{i,j}$. We use $b_{i,j}$ denotes the bandwidth of $e_{i,j}$.
- Req_k, $\text{size}^{\uparrow}(s_k)$, $\text{size}^{\downarrow}(s_k)$. We use Req_k to denote the set of requests of s_k; Others represent the attributes of s_k, including incoming and returning data size.
- $\delta(s_k)$, $\text{pri}(s_k)$, $\text{rep}(s_k)$. We use them to denote the offloading indicator, priority, replica number of s_k.

- m, C_{ik}, c_{ikm}, $\mathrm{Q}(c_{ikm})$, $\mathrm{q}(c_{ikm})$. We use C_{ik} to denote the set of containers of s_k on node i and we have $\sum_{i=1}^{|V|} |\mathrm{C}_{ik}| = \mathrm{rep}(s_k)$ for $\forall k \in Set(|S|)$. Here, we use $Set(N)$ to denote $\{1, ..., N\}$. c_{ikm} is a specific container that belongs to C_{ik}, where $m \in Set(|\mathrm{C}_{ik}|)$. $\mathrm{Q}(c_{ikm})$ is the set of resource quota that is assigned to c_{ikm}. $\mathrm{q}(c_{ikm})$ is a specific resource quota of c_{ikm}, e.g., CPU, memory.
- l, r_{kl}. We use r_{kl} to denote a request from R_k.

We model the cloud-edge-end network as an undirected graph. Nodes consist of masters and workers, which are both available to deploy services. Masters also take charge of making scheduling decisions. Each worker is connected to a master. A request is sent to a master node and then dispatched to suitable worker nodes. Nodes are equipped with heterogeneous resources, which vary with time because of resource contention. The cloud can be seen as a special worker node that is assumed to have unlimited resources.

We assume that edge services reside in one container but can have multiple replicas. Each replica has a default resource quota. For service requests, we assume that they have roughly the same data sizes and thus have similar resource requirements. Each request of a service share the same deadline $\mathcal{T}_k$, say r_{kl} arrives at the system at time t, if its processing results return to the sender before time $t + \mathcal{T}_k$, it meets the deadline; otherwise, it misses the deadline.

2.2 Problem Formulation

In containerized edge computing environment, the request latency consists of five parts, i.e., uploading $\mathcal{L}_{kl}^{\uparrow}$, transferring $\mathcal{L}_{kl}^{tran}$, waiting $\mathcal{L}_{kl}^{wait}$, processing $\mathcal{L}_{kl}^{proc}$, and downloading $\mathcal{L}_{kl}^{\downarrow}$ latency. Assuming that the master nodes forward the requests without buffering, we can describe the latency $\mathcal{L}_{kl}$ as follows:

$$\mathcal{L}_{kl} = \mathcal{L}_{kl}^{\uparrow} + \mathcal{L}_{kl}^{tran} + \mathcal{L}_{kl}^{wait} + \mathcal{L}_{kl}^{proc} + \mathcal{L}_{kl}^{\downarrow}. \tag{1}$$

For a specific request that is generated at worker v_i, transferred to master v_{j1}, and executed at worker v_{j2}, we have:

$$\begin{aligned} \mathcal{L}_{kl}^{\uparrow} &= \frac{size^{\uparrow}(s_k)}{b_{i,j1}}, \mathcal{L}_{kl}^{tran} = \frac{size^{\uparrow}(s_k)}{b_{j1,j2}}, \mathcal{L}_{kl}^{\downarrow} = \frac{size^{\downarrow}(s_k)}{b_{j2,i}}, \\ \mathcal{L}_{kl}^{proc} &= latency(sp(r_{kl}), sp(s_k), sp(v_{j2}), Q(c_m)), \\ \mathcal{L}_{kl}^{wait} &= \sum_{r \in \mathbf{Q}_{jk}} latency(sp(r), sp(s_k), sp(v_j), Q(c_{m'})). \end{aligned} \tag{2}$$

Here, $sp(\cdot)$ returns the specifications of a request, a service, or a node; $latency(\cdot)$ represents the latency model that maps resource quota to the estimated latency; $Q(\cdot)$ denotes the allocated resource quota. $\mathbf{Q}_{jk}$ represents the waiting queue on node j for service k. The details of latency modeling will be described in §3.1.

Our goal is to minimize the Deadline Miss Ratio ($\mathcal{R}$) of services. We define $\mathcal{R}$ for s_k as follows:

$$\mathcal{R}_k = \sum_{l=1}^{|Req_k|} \frac{|\{\mathcal{L}_{kl} \mid \mathcal{L}_{kl} \geq \mathcal{T}_k\}|}{|\{\mathcal{L}_{kl}\}|}. \tag{3}$$

We then draw up following execution and scaling policies:

$$\begin{aligned} \Delta &= \{\delta_{kl} \mid \forall k \in Set(|S|), \forall l \in Set(|Req_k|)\}, \\ \Lambda &= \{\lambda_{kl} \mid \forall k \in Set(|S|), \forall l \in Set(|Req_k|)\}. \end{aligned} \tag{4}$$

δ_{kl} is a variable that indicates if r_{kl} will be forwarded to execution ($\delta_{kl} = 1$) or not ($\delta_{kl} = 0$). λ_{kl} is the scaling plan of r_{kl}, which contains two parameters: (1) the replica number of s_k on each node $|C_{ik}|$; (2) the resource quota of containers on each node $\mathrm{Q}(c_{ikm})$.

With the above notations, we can formulate the problem as the following optimization problem.
Find the values of $\Delta,\ \Lambda$.

$$\mathcal{P}\ :\ \min_{\Delta,\Lambda}\ \sum_{k=1}^{|S|} \mu_k \cdot \mathcal{R}_k^{pri(s_k)} \tag{5}$$
$$s.t.\ C_1 : \delta_{kl} = \{0,1\}, \forall k \in Set(|S|), \forall l \in Set(|Req_k|)$$
$$C_2 : \sum_{k=1}^{|S|} |C_{ik}| \cdot Q(c_{ikm}) \le R_i^t, \forall i \in |V|, \forall t \in Set(\infty)$$

The weight μ_k reflects the users' preferences towards the relative importance for services with different priorities. It offers a control knob to be tuned by edge operators or users to optimize different applications and scenarios. In particular, we can decrease μ to ensure the faster response of higher-priority services or vice versa. Constraint C_1 indicates that a service request can either be forwarded to execution or on-hold. Constraint C_2 ensures that the computation resources can not exceed the resource capacity R_i^t for v_i at time t.

The major challenge that impedes the derivation of the optimal solution to the above problem is the lack of future information. To optimally solve $\mathcal{P}$ (Eq. 5), near-future information such as service workload and the estimated completion time of a service request is required. Such information is difficult to predict in advance accurately. Moreover, $\mathcal{P}$ (Eq. 5) is a mixed-integer nonlinear programming problem which is quite difficult to solve even if the future information is known as a priori. As a result, we need a hybrid approach that can efficiently adapt online conditions, make execution and scaling policies.

3 EVENT-DRIVEN LIGHTWEIGHT SCHEDULING

In this section, we would like to develop a lightweight algorithm to deal with the challenges mentioned in previous sections. We first build offline models to provide near-future information. We then develop a priority-based scheduling algorithm to give quick decisions for each request. Finally, we discuss some practical issues.

3.1 Offline Modeling

EdgeMan currently provides workload and latency models by default. Edge devices that are capable of running containers are usually equipped with direct power supplies so we do not consider the energy model.

3.1.1 Workload Model. The characteristics of containerized edge services can be different [45], and so do their requests. Some services have a high request rate within seconds [50] while others receive few requests across hours [66]. Such difference will definitely affect the accuracy of history-based models for the lack of data. To obtain more precise workload models, we first classify them into long- and short-term.

(1) For long-term services: We propose a distribution fitting mechanism with incremental refinement. Concretely, when a new service is tagged long-term (with few data points), we run a quick fitting through common distributions and choose the best one. As the requests keep coming, EdgeMan will choose to run a finer-grained fitting locally or on the cloud. The refinement process is executed periodically.

(2) For short-term services: Existing literature has explored well-established methods, such as regression [26, 53], autoregressive models [33, 80], and neural networks [33, 78]. The challenge of adopting these methods in CESs is how to address the model execution time scale and error rate trade-off gracefully. To tackle the problem, we implement three workload models, including (1) a naive statistical model that takes the max value of several prior samples as the predicted workload; (2) an ARIMA model; (3) a vanilla LSTM. At runtime, EdgeMan can dynamically trade accuracy for latency or vice-versa via model switching.

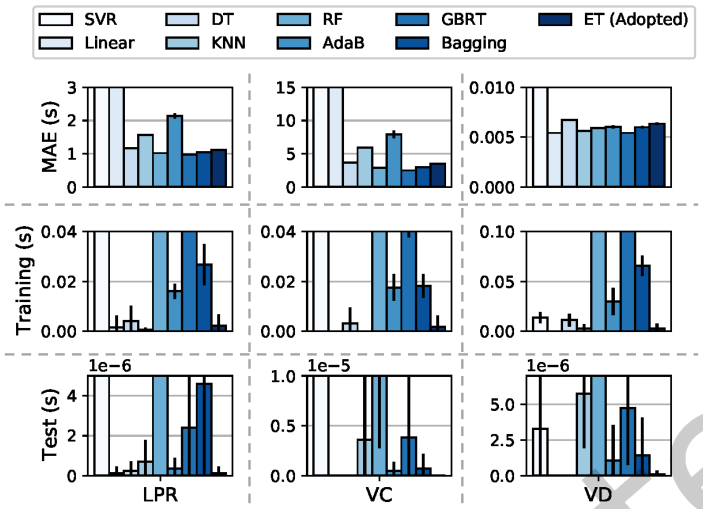


Fig. 1. Latency modeling methods comparison, including support vector regression (SVR), decision tree (DT), random forest (RF), gradient boosting regression tree (GBDT), extra-tree (ET), linear regression (Linear), k-nearest neighbor (KNN), Adaboost (AdaB), and bagging.

3.1.2 Latency Model. Ideally, we can build a white-box latency model in accordance with Eq. 1 and 2. For example, we can leverage a Petri-Net-based model [45] to characterize the lifecycle of a containerized edge service request and take advantage of queuing theory to model various waiting times [78]. Concretely, developers can tune the end-to-end latency by manipulating different parts of the latency in accordance with the real-world settings, e.g., network conditions, workload patterns, and service specifications. Unfortunately, the white-box-based methods usually have strong assumptions about the environment that are hardly applicable in highly dynamic edge scenarios. Furthermore, it is hard to model the complex lifecycle of a request precisely. So we uses black-box-based methods to estimate the request latency.

EdgeMan considers the following features for latency models: (1) the characteristics of containerized edge services, e.g., replica number and workload; (2) the runtime scenario of a request, e.g., network condition and resource consumption; and (3) computing power of a hardware platform, e.g., the frequency, time slice and other specs of CPU. After selecting the features, we evaluate a wide range of estimation methods from regression, ensemble- and deep-learning-based models in terms of mean absolute error, training, and inference time (as shown in Fig. 1). In our implementation, we use extra-tree regression as EdgeMan default solution. While EdgeMan will also periodically renew or re-select the latency model for a service.

The workload and latency models are trained for each service. The datasets are collected via a few rounds of offline execution under different settings (replica number, resource quota, etc.). When hosting a new service, we need to repeat the above procedure, which might be time-consuming. We propose corresponding techniques to reduce the cold-start time (detailed in §4.1).

3.2 Online Scheduling

3.2.1 Scheduling Actions and Mode. Given the offline models, we can try to decide on how a coming request is executed, i.e., determine Δ and Λ as defined in Eq. 4. Towards this, we comprehensively survey existing literature on container scheduling over three related topics: (1) cloud-edge system [28, 68, 74, 76, 79]; (2) Serverless

Table 2. Scheduling action comparison between EdgeMan and existing solutions. ★: Note that there are some open-source projects such as kube-batch [37] and Volcano [24] can enable batch job execution on K8s.

Category	Service-level					Request-level			
Action	Assign	Resource Allocation				Forward	Offload	Batch	Discard
		Hold	Scale Replica	Scale CPU	Scale Memory				
Concordia [23]	×	√	×	√	×	√	×	×	×
Sinan [79]	×	√	×	√	×	√	×	×	×
Metis [68]	√	√	×	×	×	√	×	×	×
KaiS [28]	√	√	×	×	×	√	√	×	×
KEIDS [34]	√	√	×	×	×	√	×	×	×
MArk [78]	√	√	√	×	×	√	×	×	√
NetMARKS [74]	√	√	×	×	×	√	×	×	×
Microscaler [76]	×	√	√	×	×	√	×	×	×
BATCH [6]	×	√	×	×	×	√	×	√	×
Fifer [27]	√	√	√	×	×	√	√	√	×
Native K8s	√	√	√	√	√	√	√	×★	×
EdgeMan	√	√	√	√	√	√	√	√	√

computing [6, 78]; (3) 5G and IIoT [23, 34] and summarize their scheduling actions in Table. 2. We found that most of the existing works focus on service-level scheduling instead of request-level, which can hardly support real-time execution.

EdgeMan embraces the merits of SOTA arts and provides a large spectrum of scheduling actions, especially for CESs. Concretely, EdgeMan can:

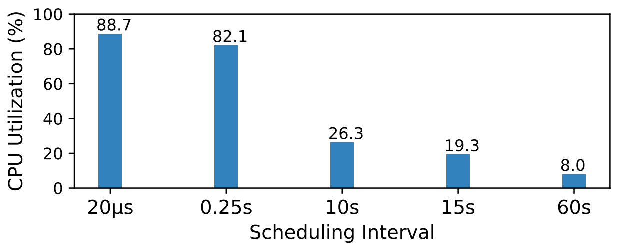
- *Assign* a service to different worker nodes (including the cloud, which is treated as a special worker node).
- *Hold* the resource allocation of a service as before.
- *Scale* up/down of a service, i.e., increase/decrease the replica number, CPU, or memory quota of a service.
- *Forward* a request to execution.
- *Offload* a request to another node.
- *Batch* a request into a waiting queue.
- *Discard* a request when it can never meet its deadline.

Another important aspect of a scheduler is the working mode, i.e., periodic or event-driven. Most schedulers mentioned above work periodically with variant time intervals from 20μs to 1h. However, scheduling services/requests with a fixed interval can either be resource unfriendly (interval is too small) or causes high deadline miss ratios (interval is too large). What is worse, even if we manage to find the magic value of scheduling intervals empirically, it is still problematic when adding new services with different request patterns. So, we build a scheduler that works in an event-driven mode. We carry out two simple simulations to further justify our claim.

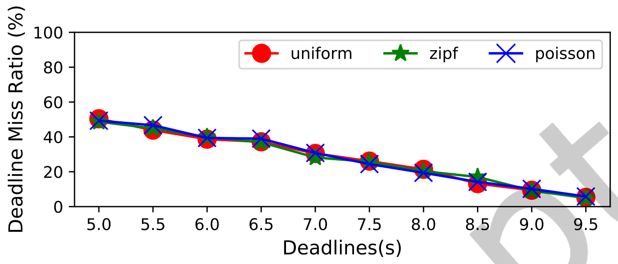
We simulate a scheduler that composes three key logic components: (1) a scraper that grabs and parses current resource statistics of worker nodes and requirements of requests; (2) a model executor that executes latency models for different services; (3) a decision maker that calls heuristic algorithm solvers to generate execution plans for requests.

Firstly, we start a daemon process that is constantly monitoring the average CPU utilization of the simulated scheduler for 10 minutes with different scheduling intervals[2]. We generate 10 requests in the queue at the beginning of every interval, each of which belongs to a distinct service. The decision maker uses random latency models (as in §3.1) and heuristic solvers (as in §4.2) to make up decision plans for the 10 requests. Fig. 2 (a) shows

[2]20 μs [23], 0.25s [28], 10s [76], 15s [34, 74], 60s [78]



(a) Simulation on scheduling interval v.s. CPU utilization



(b) Simulation on deadline miss ratio v.s. deadlines

Fig. 2. Illustrative simulation results

the results. We can see that as the scheduling interval becomes shorter, the average CPU utilization becomes larger.

Secondly, we generate 10 services with distinct deadlines, randomly inject 10k requests (1k for each service) to a fix-interval scheduler. Requests follow different distributions (uniform, zipf, and poisson in this simulation). The deadlines are uniformly sampled around its interval at a step of 5%, i.e., let the interval equals to 10s, then the deadlines for 10 services are: 5s, 5.5s, ..., 9s, 9.5s. Assume that each request will finish immediately once scheduled (i.e., with zero execution time). We measure the average deadline miss ratio. Fig. 2 (b) shows the results. We can see that a fix-interval scheduler itself can cause high deadline miss ratio (up to 50%) regardless of models or scheduling algorithms.

Although Metis scheduler works in the event-driven mode, it is designed for long-running applications and its scheduling decision time will last tens of minutes [68], which is not suitable for our scenario. As a result, EdgeMan builds a scheduler that works in an event-driven mode but proposes scheduling algorithms on its own.

3.2.2 Priority-based Fair Scheduling Algorithm. When multiple requests of different services arrive simultaneously, EdgeMan will decide the resource allocation sequence for each service, i.e., the execution plan for the corresponding requests based on the service priority. Moreover, considering the distinct characteristics of long- and short-term services, EdgeMan borrows the idea of Jain's fairness index [32] in network engineering to balance the varying workload features for services with the same priority. Concretely, we redefine the fairness index as follows:

$$\mathcal{J}(x_1, x_2, ..., x_{|S|}) = \frac{(\sum_{i=1}^{|S|} x_k)^2}{n \cdot \sum_{i=1}^{|S|} (x_k)^2} = \frac{\bar{x}^2}{\bar{x^2}}, \tag{6}$$

where x_k is the proportional workload for s_k, e.g., a typical x_k can be defined as follows:

$$\frac{req\_num\ in\ recent\ 1\ minute}{estimated.\ req\_num\ in\ recent\ 1\ hour}. \tag{7}$$

To achieve a given fairness level F, one approximate method is to let $x_k = A \cdot k^{\alpha}$, where $\alpha = \frac{1-F+\sqrt{1-F}}{F}$ and A is an arbitrary factor, typically used for normalization.

Algorithm 1 Priority-based fair scheduling algorithm

```
Input: Request list $req\_list_t$, fairness level $F$, arbitrary factor $A$, specs of services and nodes $specs$, service models $M$, current
    timestamp $t$, service deadlines $\mathcal{T}$.
Output: Allocation sequence $seq_t$, execution plan $p_t$.
1: $req\_groups \leftarrow$ group($req\_list_t$, key=$specs$.s.priority)
2: for $g \in req\_groups$ do
3:     index $\leftarrow$ cal_index($g$, $F$, $A$)
4:     $g \leftarrow$ sort($g$, key=index)
5:     $seq_t \leftarrow seq_t \cup$ {g}
6: for $seq \in seq_t$ do
7:     if not service_is_assigned($seq$) then
8:         $p_t^{seq} \leftarrow p_t^{seq} \cup$ {“assign”}
9:     for $req \in seq$ do
10:        if $t$ + cal_T($M$, $specs$) $\leq \mathcal{T}_{req}$ then
11:            $p_t^{seq} \leftarrow p_t^{seq} \cup$ {“forward”}
12:        else if $t$ + cal_offload_T($M$, $specs$) $\leq \mathcal{T}_{req}$ then
13:            $p_t^{seq} \leftarrow p_t^{seq} \cup$ {“offload”}
14:        else if $t$ + cal_worst_T($M$, $specs$) $\leq \mathcal{T}_{req}$ then
15:            $p_t^{seq} \leftarrow p_t^{seq} \cup$ {“batch”}
16:        else if $t$ + cal_scale_T($M$, $specs$) $\leq \mathcal{T}_{req}$ then
17:            $p_t^{seq} \leftarrow p_t^{seq} \cup$ {“scale”}
18:        else
19:            $p_t^{seq} \leftarrow p_t^{seq} \cup$ {“discard”}
20:    $p_t \leftarrow p_t \ \cup \ p_t^{seq}$
21: return $seq_t$, $p_t$
```

Algorithm 1 shows the pseudo-code of our algorithm. The input of the algorithm contains: (1) The request list at timestamp t, req_list_t. (2) The fairness level F and arbitrary factor A are used in the fairness index calculation. (3) Specifications of services and nodes $specs$ together with service models M, which estimate workload and latency. (4) Service deadlines $\mathcal{T}$. The output is the resource allocation sequence seq_t and execution plan p_t for each requests.

We first group together requests with the same priority (Line 1). Then, we calculate the fairness index for each group and sort the group accordingly (Line 2-4). We can thus aggregate the sorted group and get the allocation sequence (Line 5). After that, we iterate each request in each priority group to generate the execution plan in the sequence of forward, offload, scale, and discard (Line 6-19). Specifically, we will check if the service that the requests belong to has been assigned to a worker node (line 7-8). When a request can presumably meet its deadline, it will be forwarded directly (Line 10-11). Otherwise, we first try to offload the request because it requires no adjustment on local nodes (Line 12-13). If the current network condition is poor or the transmission data is too large, we can try to batch the request for a while, e.g., a user-defined interval, waiting for potentially

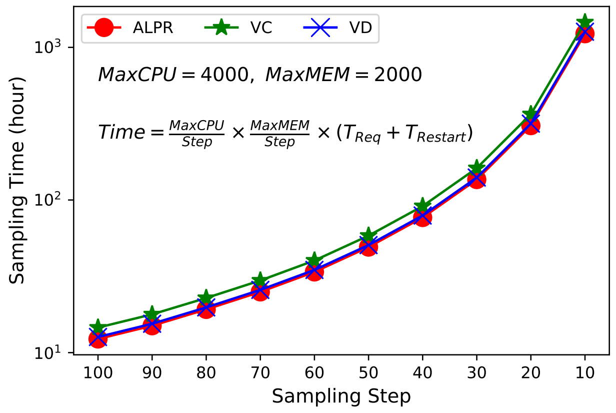


Fig. 3. Data preparation time for latency models.

less stringent context (Line 14-15). If batching is not viable, we then evaluate the possibility of scaling the service (Line 16-17). If the above methods still cannot enforce the request to meet its deadline, it will be declined (Line 19). Finally, we can aggregate and get the execution plan p_t (Line 20). EdgeMan scheduler will parse and enforce the plan thereafter.

However, there are still some practical issues when providing real-time support for containerized edge services. First, building precise offline models for each service requires comprehensive profiling, which has the cold-start problem and is time-consuming. Second, computing resource is essential, especially for those resource-constrained edge devices, how to effectively allocate just enough resource when scaling is also essential. We will further address the above issues in the next section.

4 BOTTLENECK-AWARE PROGRESSIVE RESOURCE ALLOCATION

As we have mentioned in the previous section, offline modeling is hardly applicable for its exhaustive profiling. In this section, we would like to propose a bottleneck-aware progressive resource allocation mechanism to address this.

4.1 Progressive Bottleneck Metric Modeling

To get a precise model, we have to use small sampling steps and the resulting preparation time is high. We carry out a simple simulation to illustrate the time scale. We split sampling time into two parts, the request processing time and container restart time[3]. We measure the request processing time under maximum resource quota and use that for smaller resource quota, which is a substantial underestimation. We also measure the container restart time. The simulation is run through different sampling steps. Results are shown in Fig. 3. We can see that the time scale can reach a hundred hours for one service.

Fortunately, not all sampling steps are required. For CPU-intensive services, memory or other resources may have a negligible effect on the execution efficiency and vice versa. Instead of exhaustive profiling, we propose a progressive bottleneck metric modeling method to profile relevant resources efficiently. The basic idea is to measure the gradient between latency and resources. However, such a simple model might not function well as the relationship between resource quota, and service performance can have non-linearity. Towards this, we

[3]Currently, changing the resource quota of a K8s pod will cause a mandatory restart. The community is working on an in-place resource update.

divide resources into different levels, e.g., say a service is allocated with 1 CPU core, we can divide the intervals every 0.1 and get ten gradients. After that, we merge the levels with similar values and get the final sequence. Formally, the above process is summarized in Algorithm 2.

Algorithm 2 Bottleneck metric determination algorithm

```
Input: Resource quota set $Q$, service set $S$, level intervals $Inter$, similarity factor $\epsilon_s$, convergence factor $\epsilon_c$.
Output: Bottleneck metric sequence $bseq$.
1: for $s \in S$ do
2:     for $q \in Q$ do
3:         $\text{min}_q \leftarrow$ find_Valley($s$, $q$, $\epsilon_c$, repeat=5)
4:         $\text{max}_q \leftarrow$ find_Plateau($s$, $q$, $\epsilon_c$, repeat=5)
5:         Levels $\leftarrow$ gen_Levels($\text{min}_q$, $\text{max}_q$, $\text{Inter}_q$)
6:         Levels $\leftarrow$ merge_Levels(Levels, $\epsilon_s$)
7:         for $level \in$ Levels do
8:             $grad \leftarrow$ cal_Grad($level$, repeat=3)
9:             $bseq\_s\_q\_l \leftarrow bseq\_s\_q\_l \cup \{grad\}$
10:        $bseq\_s\_q \leftarrow bseq\_s\_q \cup bseq\_q\_l$
11:    $bseq\_s \leftarrow bseq\_s \cup bseq\_s\_q$
12: $bseq\_s \leftarrow$ prune($bseq\_s$, $\epsilon_s$, primary=3)
13: $bseq \leftarrow bseq \cup bseq\_s$
14: return $bseq$
```

For a resource type of a service, we first probe the min/max values and generate levels (Lines 1-5). Then we merge levels with a predefined similarity factor ϵ_s (Line 6). Given the final levels, we iteratively calculate the gradient (Lines 7-8). We can then aggregate gradient sequence across levels and resource types (Lines 9-11). The bottleneck metrics are determined by pruning unimportant resource types (Lines 12-14).

Although faster than the brute-force method, exhaustively determining bottleneck metrics can still hardly be done in real-time. Fortunately, this method has a short start-up time. We can run a few rounds of the algorithm and get the rough bottleneck model for each resource. A background daemon can then refine and update the model progressively.

4.2 Extensible Resource Allocation Mechanism

Given the offline models (from §3.1), resource allocation sequence (from §3.2), and bottleneck metric(s) (from §4.1), we further advocate the resource allocation mechanism. The objective is to decide how many resources should be allocated for a service. The basic idea is to search the allocation space in a bottleneck-aware manner.

Considering the exponentially large space, searching for the optimal allocation is impractical. EdgeMan leverages heuristic algorithms to give a sub-optimal but acceptable solution in a short time. There are various algorithms to choose from, such as Differential Evolution (DE) [59], Genetic Algorithm (GA) [73], Simulated Annealing (SA) [64], Immune Algorithm (IA) [67], and Particle Swarm Optimization (PSO) [35], etc.

To comprehensively evaluate the efficiency of existing heuristic algorithms, we run five widely-adopted alternatives in EdgeMan testbed with sample traces and keep track of their resultant latency mean absolute error (compared with a user-defined deadline), algorithm processing time, and the memory footprint. As shown in Table. 3, PSO prevails other algorithms in terms of MAE, processing time, and memory footprint. So we choose to use PSO as the decision-maker.

With the help of bottleneck metric, EdgeMan can further speed up the searching procedure. We can fix resource types that are not included in bottleneck metrics with a slightly larger value than the minimum. Such

Table 3. Heuristic algorithm comparison

Algorithms	MAE	Processing Time	Memory
DE	0.00704s	0.0352s	0.434MB
GA	0.00706s	0.1111s	0.344MB
SA	14.85270s	0.2353s	0.008MB
IA	128.37839s	0.1892s	0.371MB
PSO	0.00704s	0.0193s	0.016MB

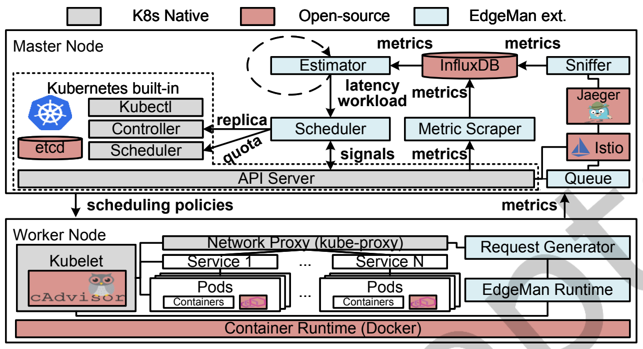


Fig. 4. Architecture of EDGEMAN testbed prototype.

a method dramatically reduces the dimension of the search space. Also, we can greedily search the bottleneck metrics from the most important to the least, which can be interrupted but still give good results.

5 IMPLEMENTATION

In this section, we will briefly introduce K8s and the key features relevant to this work, followed by an overview of the proposed extended components. Fig. 4 shows the overall system architecture of K8s that integrates EDGEMAN.

1) *K8s:* K8s is a framework designed to orchestrate containerized workloads on clusters. The basic building block is a pod, which encapsulates one or more tightly coupled co-located containers that share the same set of resources. The amount of requested and/or limited resources (including CPU, memory, storage, etc.) is suggested to be specified for a pod in order to make reliable decisions on pod placement (*scheduler*). A pod is designed to run a single instance of an application. As a result, multiple pods can be used to horizontally scale an application (*controller*). K8s *API server* validates and configures data for the API objects. *Kubelet* is the primary "node agent" that can register nodes with the *API server*, and monitoring the running status of a pod, where *cAdvisor* provides an understanding of the resource usage and performance characteristics.

2) *Extended Components:* On master nodes, the *Istio controller* will take over all the requests of registered services into *request queue* and waiting for the execution plan of *EDGEMAN scheduler*. More specifically, an Istio sidecar (i.e., an Envoy proxy) will be injected into each service for request routing. EDGEMAN also integrates Jaeger, a distributed tracing tool that can grape the request route and latency. We implement a python gRPC client that will grab tracing results from Jaeger. The *metric aggregator* will continuously monitor the state of nodes and pods in the cluster, including current replica number, resource consumption, and network conditions. The collected data are formalized and stored in a time-series database *InfluxDB*, which is a popular open-source solution in time-series data storage. Given the formalized metrics, the *estimator* makes predictions not only on the coming workload of services requests but the completion time of the requests, which enables the *EDGEMAN scheduler* to

draw up proper scheduling policies. On worker nodes, we build a *request generator* module that will generate requests when the system has idle resources for progressive model refinement. *EdgeMan runtime* continuously fetches metrics from *kubelet* and transmits them to the master nodes.

As mentioned in §3, *EdgeMan scheduler* works in an event-driven manner. More specifically, the scheduler will be triggered by the following three types of built-in signals:

(1) *request-arrival*: whenever a new request arrives.
(2) *resource-changed*: when the typical resource (e.g., CPU, memory, bandwidth, etc.) changes greatly (determined by a user-defined threshold).
(3) *time-expired*: users can define timers that enforce EdgeMan scheduler work periodically; whenever a timer expires, the signal will trigger the scheduler.

Note that users can also define new signals for EdgeMan scheduler, e.g., *request-finished.* Due to the non-in-place nature[3], request-triggered resource reallocation is currently not realistic. Towards this, EdgeMan provides two alternatives. One and the default version is K8s native, i.e., periodically sync and change the resource allocation. The other is the user-customizable trigger. For example, users can define a trigger as "IF $\mathcal{R}_k^{pri(s_k)} > threshold\ in\ the\ recent\ 10\ minutes$ THEN enforce resource reallocation".

Results of *EdgeMan scheduler* includes service- and request-level. The former consists of the service assignment plan, the replica number, and the resource quota. The latter consists of the request execution plan i.e., to forward, offload, batch or discard the request. The results will be sent to corresponding modules that take charge of the execution.

6 EVALUATION

Our evaluation tries to answer the following two questions:

How does EdgeMan perform in practice? We build a three-service testbed and generate requests according to realistic situations. Testbed experiments (Fig. 7, 8, and 11 (a)) show that EdgeMan can achieve efficient request-level scheduling and resource allocation with low overhead.

How effective is EdgeMan in large-scale scenario? Large-scale simulations (Fig. 5, 6, 9, 10, 11 (b), and (c)) show that EdgeMan workload model can cope with highly dynamic situations.

6.1 Services and The Host Machine

We use the following three time-sensitive services throughout the experiments. The LPR service recognizes the license plate number from a fix-size image. We use the widely deployed OpenALPR [51] as the service backend. The VC service counts the number of vehicles from a fix-size video. We use TensorFlow object counting API [5] to implement the service backend. The VD service classifies a fix-size time-serial accelerometer data into five different levels of vibration. We use a simulated KNN-based implementation [18]. We set up RESTful API servers for these services and pack them up into K8s pods.

To host the above services, we use a single PC as both the master and worker. It is equipped with an Intel Core i7-8700 CPU @3.2GHz and 32 GB memory. The following real-world and simulation experiments are carried out on the PC. Our testbed is only an illustrative implementation of our design. We believe that such a single PC setup for holding three single-module services and proving that the effectiveness of EdgeMan is enough. In the future, we will adapt our solution to more complex situations such as services with multiple modules and use a large-scale cluster to evaluate.

6.2 Baselines

EdgeMan includes a scheduling algorithm and a resource allocation mechanism.

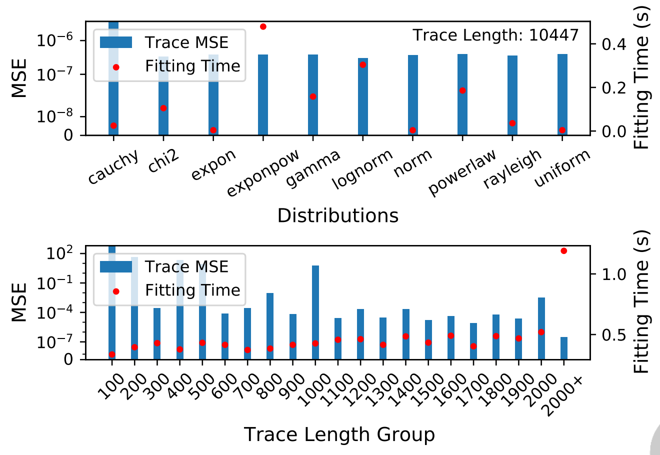


Fig. 5. Snapshots of the Alibaba trace with fitting results and overhead.

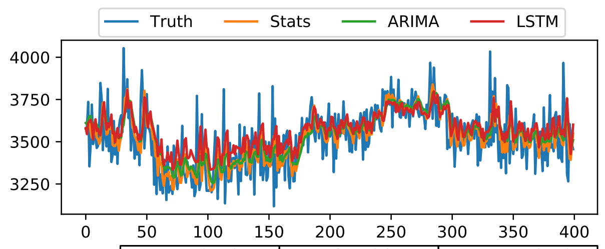
6.2.1 Scheduling Algorithm Baselines. We compare our solution with two K8s baselines: HPA and VPA. Specifically, HPA scales services horizontally, i.e., adding/reducing the number of replicas in accordance with predefined CPU and memory threshold, which is set to 80%. VPA scales services vertically, i.e., adding/reducing the resource quota based on usage history. We also include SOTA works in related areas. Specifically, we group them into two categories: (1) Only scale CPU cores [23, 79] (referred to as SCPU). (2) Only scale service replicas [30, 76, 78] (referred to as SREP). Note that most existing schedulers work in a periodic manner, to compare fairly, we use three scheduling intervals that are closer to the three services we evaluated: 20μs [23], 0.25s [79], and 1s [28]. In the following experiments, SCPU and SPRE will be evaluated under these intervals and we keep the best results.

6.2.2 Resource Allocation Mechanism Baselines. To compare resource allocation mechanisms. Similarly, we use HPA, VPA, SCPU, and SREP as baselines. HPA will increase/decrease resource allocation in the unit of service, e.g., say a service is scaled up from 1 to 2, the resource will be doubled. VPA will increase/decrease resource allocation in a more fine-grained way, i.e., a small portion of CPU or memory for all replicas. SCPU only increases/decreases the CPU quota in the unit of cores. Note that SREP, in terms of scheduling or resource allocation, is exactly the same as HPA, except that SREP uses the same workload and latency estimation/trigger from EdgeMan.

6.3 Main Results

6.3.1 Workload Prediction. As there are no publicly available traces for our services, we synthesize service workload based on Twitter [56], and Alibaba [7] traces. We believe that Alibaba traces can provide real-world request patterns for long-term container-based services. In contrast, Twitter traces serve as a good micro-benchmark for short-term service as it has highly dynamic workloads.

We group Alibaba traces according to trace length and evaluate the performance of different distributions for one trace and the overall performance across traces. Results (Fig. 5) show that the long-term workload models can quickly converge with a reasonable number of data points. On the other hand, we use 20 days of Twitter traces within a month to train short-term workload models while the rest for testing. Fig. 6 illustrates a snapshot of the trace along with the prediction results and the overhead. The overall results show that the prediction accuracy is good enough for the Twitter trace, which means EdgeMan short-term workload model can handle complex and dynamic circumstances.



	MSE (req)	Time (s)	Memory (MB)
Stats	22476	1e-09	1e-09
ARIMA	19912	0.00211	26.999
LSTM	16872	0.00482	777.817

Fig. 6. Snapshots of the Twitter trace with prediction results and overhead.

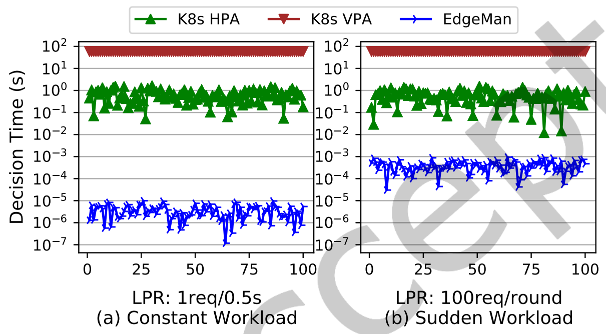


Fig. 7. The scheduling efficiency comparison.

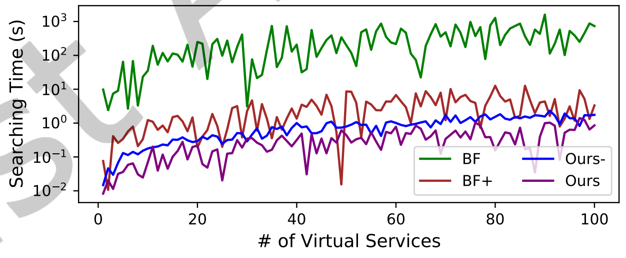


Fig. 8. Bottleneck metric searching time comparison among brute force (BF), brute force with bottleneck metric awareness (BF), and our method with (Ours) or without bottleneck metric awareness (Ours-).

6.3.2 Scheduling Efficiency. To evaluate the scheduling efficiency, we generate two types of workloads, i.e., the constant (or periodical) and sudden workload for the LPR service. We monitor and measure the scheduling policy generation time (not including the time of enforcing the policy) of HPA, VPA, and EdgeMan for 100 times. We do not include SCPU and SREP here as they work in a similar periodical manner with HPA and VPA. Results are plotted in Fig. 9-10. We can see that K8s HPA and VPA are not sensitive to the workload type; they just constantly grape metrics and generate decisions in a request-unaware manner. While EdgeMan will react to each request,

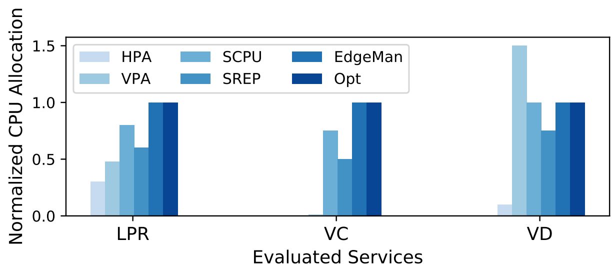


Fig. 9. The CPU allocation efficiency comparison.

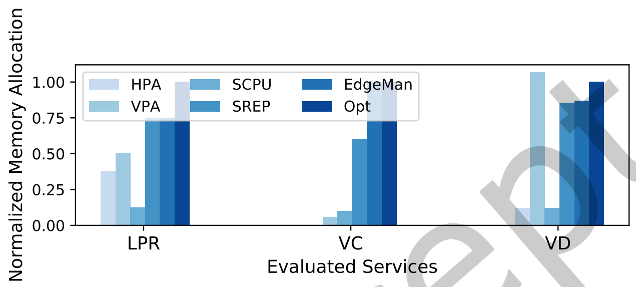


Fig. 10. The memory allocation efficiency comparison.

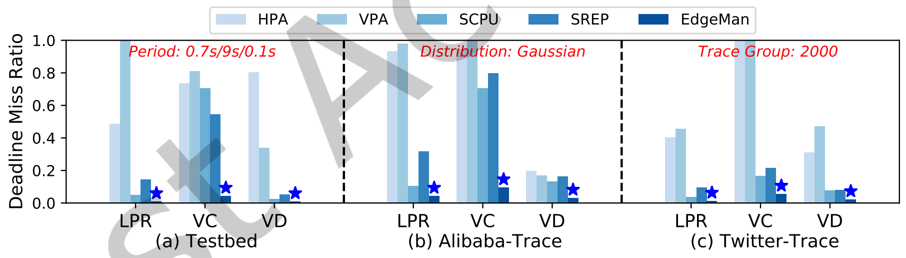


Fig. 11. Deadline Miss Ratio comparison of K8s and EdgeMan.

which suffers performance degradation as the number of requests increases. Such degradation is still acceptable when the *concurrent* workload is less than 100,000.

6.3.3 Resource Allocation Performance. We evaluate the allocation performance in terms of latency and resource efficiency. (1) We measure the time of generating a resource allocation plan for four methods: Brute forth (BF), BF with bottleneck metric awareness (BF+), our method without bottleneck metric awareness (Ours-), and our method (Ours). We simulate a number of virtual services and use the same step to search the whole resource allocation space. (2) We generate and inject arbitrary requests for the three services and empirically obtain the optimal resource allocation (marked as Opt in Fig. 9 and Fig. 10). Then we measure testbed resource allocation results for EdgeMan and other baselines.

For allocation plan generation (Fig. 8), the brute-forth method, even with bottleneck metric awareness, still takes a longer time than our method. For allocation efficiency, we plot the optimal resource as blue bars in Fig. 9 and Fig. 10. We can see that existing solutions tend to give under- or over-estimation results while EdgeMan is obviously more resource-efficient, saving 76.79% of CPU and 1.18x of memory resources on average compared to baselines.

The main reasons that EdgeMan can perform better than baseline are two-fold: (1) Our scheduler works in an event-driven and workload-aware manner, which is more reactive than that of periodical scheduling. Specifically, for computation-intensive services (e.g., VC), HPA and VPA may even not be able to give resource allocation before the service is crushed by requests. (2) EdgeMan builds bottleneck metric models for each service, which therefore can help to allocate resources in a well-targeted way with higher efficiency.

6.3.4 End-to-End Deadline Miss Ratio. We design three scenarios to evaluate our solution in periodic and dynamic real-world traces with distinct patterns. Specifically, we (1) adopt the arbitrary testbed settings and send *periodical* requests for each service (0.7s/req for LPR, 9s/req for VC, 0.1s/req for VD); (2) use Alibaba traces to generate *long-term* requests that follow the Gaussian distribution; (3) regroup Twitter traces according to the total number of requests in 1 minute and choose a representative trace group 2000 (i.e., there are more than 2k requests within 1 minute); we arbitrarily sample the request arrival time in the group and form a sequence. The original request pattern of both Twitter and Alibaba is more suitable for the cloud or datacenter setting. To make it more suitable for the edge scenario, we have adjusted the workload in accordance with the hardware performance. For example, we leverage the information provided in the dataset schema of the Alibaba trace to tune the workload. Specifically, in *machine_meta.csv* and *machine_usage.csv*, we can know the number of CPU (*cpu_num*) and memory size (*mem_size*) on a typical machine (*machine_id*). Given the information, we can then adjust the workload by multiplying a ratio, which typically can be defined as: $ratio = cpu\_num\_of\_testbed\_PC/cpu\_num\_of\_dataset\_machine$. On the other hand, the scheduling intervals are set to default for HPA and VPA (i.e., 15s). While to strike a fair comparison with academic works, we use a minimal interval (20μs [23] in our experiment) and use our workload and latency models for SCPU and SREP. We then send the generated requests and record the deadline miss ratio.

Fig. 11 shows the results. We can see that: (1) For periodical and short-term requests, existing works tend to perform poorly due to the long sync period and request unawareness (Fig. 11 (a) and (c)). (2) For long-term requests, existing works can have a fair performance (Fig. 11 (b)). While we can do well in all three scenarios due to the fast decision-making and the request-level control loop. The overall deadline miss ratio is reduced 85.9% on average.

6.4 Overhead

Table. 4 shows the overhead comparison among EdgeMan, Microscaler [76], and MArk [78]. The *metric aggregator* module takes 0.7 ∼ 2% single CPU utilization when collecting model input of *estimator*. To give accurate latency and workload prediction results, EdgeMan consumes less than 0.01 and 0.1 seconds with little memory footprint. In *EdgeMan scheduler* module, it takes less than 0.1 seconds to generate multidimensional scheduling policies for one service; while Microscaler and Mark require much higher resources. The main reasons that EdgeMan can achieve such efficiency are that: we (1) adopt fast latency models; (2) use different workload models to cope with resource budget; (3) the scheduling and allocation mechanism is lightweight.

7 DISCUSSION

In this section, we will discuss future research directions or improvements towards EdgeMan.

7.1 Assumption Relaxation

Table 4. Overhead comparison

Functionality	**Overhead**		
	EdgeMan	Microscaler	MArk
Data Collection	0.7 ∼ 2%	5 ± 1%	≥ 2%
Violation Detection	<0.01 sec	∼0.2 sec	2∼10 min
Schedule Decision	<0.1 sec	∼2 min	

In §2 we have made the following assumptions to simplify the real-time support problem:

(1) The end devices are connected to the edge network via wired cables.
(2) The edge service resides in one container.
(3) The requests of a service share the same data sizes.
(4) The edge devices have direct power supply.

These assumptions might not hold in other cases and thus will need to be relaxed.

For assumption (1): Although there are many edge scenarios that the end devices are indeed connected to the edge networks via wired cables (e.g., structural health monitoring [15] and natural hazard monitoring [47]), there are also scenarios that require wireless connection from end devices to the edge (e.g., building-centralized system [72] and video interpretation [75]). In these scenarios, the network conditions might not be so stable as wired connection. Fortunately, there are many research works on throughput or bandwidth prediction [46, 48, 77] for all kinds of transmission protocols in the literature. Users can build and integrate network models seamlessly in EdgeManas workload and latency models.

For assumption (2): Some edge devices have limited computing resources and thus are capable of holding small-scale edge services, which can reside in one single container. There are, however, some other edge devices are equipped with abundant resources, e.g., edge servers. It is beneficial to evaluate EdgeMan in more complex scenarios, e.g., the containerized edge services are composed of many micro-services or the number of containerized edge services is large. In such scenarios, a larger scale K8s cluster (e.g., 1 master + 9 workers) may be necessary. Unfortunately, there are limited real-world edge benchmarks that are suitable for such complex scenarios according to the survey conducted by Varghese et al. [65]. So, we instead run large-scale simulations using traces provided by Alibaba cloud and Twitter (see Fig. 11).

For assumption (3): In reality, it is quite hard to estimate how many data a request carries. We can try using history-based methods to predict or just set the deadline of that service to the worst case (i.e., latency required to process the empirically maximum data sizes).

For assumption (4): There are also cases that energy consumption matters even with a direct power supply, e.g., in 4G/5G cell towers or large-scale edge clusters. Similar to assumption (1), there are also works make effort to save energy for edge computing systems [27, 34, 62]. EdgeMan can enable energy-aware scheduling by integrating energy profiles and models of worker nodes and service requirements.

7.2 Work in Other Settings

7.2.1 Deadline Settings. In our setting, it is the service users who define the sensible threshold critical to maintain quality of service. One way to set the deadline is to multiply the one-shot execution time (the smallest average execution time of this service on this edge node) to a ratio that is a typical proportion, e.g., 10%. As EdgeMan focuses on reducing the end-to-end deadline miss ratio for time-sensitive services, so we set a relative stringent deadline for each of them (e.g., 5% for VC), just for illustration. It is also worth noting that the deadline settings for services do not affect the effectiveness of EdgeMan.

7.2.2 Long Services. It is true that edge system operator should make long services (e.g., 10 secs such as VC) to be served as asynchronous task, which in turn requires users to constantly polling the service results of the requests instead of using synchronous API calls. However, for end users who actually call the service, no matter it is a synchronous or asynchronous call, what users care about is the perceived end-to-end latency. It is possible that EdgeMan latency model will suffer from a performance degradation for services that are called in an asynchronous manner (e.g., because of the extra overhead of database access), we argue that we can make compensation for the original models or design typical features for this situation, and we leave it for future exploration. With proper modifications of the latency model, we believe that EdgeMan can also work well for these services.

7.2.3 Running and Evaluating EdgeMan over a Larger-Scale Testbed. As we mentioned in §6.1, our testbed is only an illustrative implementation of our design, and we plan to adapt our solution to more complex situations such as services with multiple modules and use a large-scale cluster to evaluate. We believe that EdgeMan can work seamlessly in complex scenarios that require larger scale K8s cluster for the following three reasons: (i) The K8s infrastructure EdgeMan runs on top of can be naturally extended to more workers and cloud testbeds such as CloudLab [20] can provide enough hardware resources to operate. (ii) In such cases, services are usually spread out in many containers and current latency models might not be so accurate as before. One naive solution is to build inter-communication models one-by-one for the service. Or we can extract features from service topology and build new latency models like Sinan [79]. Moreover, if the containerized edge services are composed of multiple micro-services, we can tune our latency model like Kraken [12] and StepConf [71], which take explicit consideration of multiple micro-services to yield better modeling performance. (iii) When dealing with scenarios with multiple micro-services, it is also beneficial to take explicit consideration of cross-module or cross-machine interactions. We can update EdgeMan scheduler by leveraging the scheduler plugin of K8s to implement new scheduling policies as in NetMARKS [74] and Beam [57].

8 RELATED WORK

In the literature, there is a large body of research on performance modeling and service scheduling. Here, we introduce relevant recent works in containerized edge computing scenario and discuss the difference with EdgeMan.

8.1 Workload and Latency Modeling

MobiQoR [38] models the service response time and app energy consumption in a white-box manner, considering features like time of data transfer, task processing, and power. But MobiQoR does not take the effect of multi-thread execution and dynamic workload into consideration. To tackle this issue, Guan et al. propose Queec [26], where they use regression techniques to model the execution time and the workload of specific edge services. In line with the above works, Loghin et al. [40] give a thorough analysis of the performance in hybrid edge-cloud processing; they build different processing time models under edge-cloud situations considering only the data size of the transmission. Other works [19, 42] leverage M/M/C queuing model and divides the overall latency into transmission delay, routing delays, and the cloud processing time. Medel et al. [43–45] use a Reference Net to model the lifecycle of a K8s pod. MArk [78] leverages a vanilla LSTM as the default workload model and uses two heuristic-based methods to capture the characteristics of ML service latency. More recently, Sinan [79] introduces a hybrid approach that integrates a multi-channel CNN and a Boosted Trees to model end-to-end latency of container-based applications.

Unfortunately, the aforementioned works cannot adapt to highly dynamical context of CESs. Main reasons are two-fold: (1) They do not consider unique characteristics (e.g., interval pattern) of workloads for CES service. (2) The proposed methods fail to achieve reactive model update because they use either heavy-weight simulation tools [43–45] or require enough data for thoroughly retraining [78, 79]. EdgeMan, on the other hand, proposes a flexible mechanism that can adapts to dynamic workload patterns by model switching. We also implement lightweight latency models based on unique features of CESs, enabling dynamic model update.

8.2 Containerized Service Scheduling

Most of existing works on containerized service scheduling focus extensively at pod level, i.e., the minimum scheduling unit is a pod. These works can be further categorized by scheduling actions.

Firstly, researchers pay their attention on pod assignment (i.e., deciding to deploy a pod on which node in the cluster). Zhang et al. [70] use native K8s pod scheduling score (i.e., priority, deciding which pod to be scheduled first) and propose an improved PSO algorithm to make the decision. Ungureanu et al. [63] propose a scheduler that encompasses both centralized and distributed functionalities to enable optimal pod placement. KEIDS [34] formulates an integer-linear-programming-based multi-objective problem to minimize energy consumption and assigns service that can have less interference with each other in IIoT scenario. While Townend et al. [62] consider the energy issue in data centers. They incorporate software with hardware models for the K8s scheduler and yield a 10-20% reduction of power consumption. Chima Ogbuachi et al. [14] look at the K8s scheduling at a real 5G infrastructure. They introduce a new priority score that considers 5G specific features to assign pods. Wang et al. [68] use reinforcement learning to properly place pods and improve the throughput performance of long-running containerized applications. Rodriguez et al. [52] make a step further. They use a best fit bin packing to assign pods and propose a simple autoscaler to decide whether to start a new node or shutdown an existing node. More recently, Wojciechowski et al. [74] integrate dynamic network metrics into K8s scheduler and improve application response time in the cross-node communication scenario, which is superior to work proposed by Santos et al. [55] that leverage static network condition, i.e., RTT, to optimize pod placement. Zhong et al. [81] advocate a heterogeneous task allocation mechanism that save the cost by task packing. El et al. [22] build a model for K8s pod lifecycle and propose a new algorithm that assign pods to heterogeneous cluster consisting of CPUs and GPUs.

Another line of works concentrate on pod auto-scaling (i.e., dynamically add or reduce the number of pod replica). Zhao et al. [80] use the ARIMA to predict workload and scale the service accordingly. KubeSphere [11], on the other hand, focus on fair resource allocation for each service. More recently, Rzadca et al. advocate Autopilot [54] that uses machine learning algorithms (e.g., exponentially-smoothed sliding window and meta-algorithm) applied to historical data to optimize both HPA and VPA, yielding better resource utilization and less OOM effects. Toka et al. [61] assume Markovian request arrival and combine queuing model with a reinforcement learning-based policies to estimate request number throughout a day and proactively scale the pods. Choi et al. propose pHPA [16], which proactively bootstrap pods for chained microservices with the help of a GNN-based model.

To provide real-time support in CESs, only pod-level scheduling is not satisfactory. Requests can still violate their deadlines without triggering any pod-level scheduler monitors. There are, however, little works pay attention to request-level scheduling for containerized services. Microscaler [76] introduces a scaling score that considers request-level features (e.g., the average latency for the slowest 10 percent) and uses Bayesian optimization to decide the number of pods. KaiS [28] proposes a learning-based scheduling framework that can maximize the long-term system throughput by dispatching requests, placing and scaling service.

Unlike the above works, EdgeMan focuses on request-level real-time scheduling for CESs. We explore broader action space, deal with unique challenges, and yield better results.

9 CONCLUSION

In this paper, we conducted a study of operating real-time CESs. We proposed EdgeMan, a holistic autoscaling solution that integrates a request-aware lightweight scheduling algorithm and a bottleneck-aware efficient resource allocation mechanism. Evaluation results show that compared with exiting solutions, EdgeMan yields significant deadline miss ratio reduction (85.9% on average) while incurs acceptable overhead. In the future, we will consider more practical CES situations such as complex services, scheduling fairness, large-scale testbed experiment, etc.

REFERENCES

[1] 2022. K8s Scheduling Framework. https://v1-23.docs.kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.
[2] 2023. Autoscaling on metrics not related to Kubernetes objects. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/#autoscaling-on-metrics-not-related-to-kubernetes-objects.
[3] 2023. Autoscaling on multiple metrics and custom metrics. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/#autoscaling-on-multiple-metrics-and-custom-metrics.
[4] 2023. WRITING EXPORTERS. https://prometheus.io/docs/instrumenting/writing_exporters/.
[5] ahmetozlu. [n.d.]. Tensorflow Object Counting API. https://github.com/ahmetozlu/tensorflow_object_counting_api.
[6] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *Proc. of ACM/IEEE SC*. 1–15.
[7] Alibaba Cloud. [n.d.]. Alibaba Clusterdata. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018.
[8] Amazon Web Services, Inc. 2023. Container benefits. https://docs.aws.amazon.com/whitepapers/latest/docker-on-aws/container-benefits.html.
[9] Azure IoT. [n.d.]. Azure IoT Edge: Cloud intelligence deployed locally on IoT edge devices. https://azure.microsoft.com/en-us/services/iot-edge/.
[10] Seyed Morteza Babamir. 2012. *Real-Time Systems, Architecture, Scheduling, and Application.* BoD–Books on Demand.
[11] Angel Beltre, Pankaj Saha, and Madhusudhan Govindaraju. 2019. Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters. In *Proc. of IEEE Cloud Summit*. 14–20.
[12] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proc. of ACM SoCC*. 153–167.
[13] Giorgio C Buttazzo. 2011. *Hard real-time computing systems: predictable scheduling algorithms and applications.* Vol. 24. Springer Science & Business Media.
[14] Michael Chima Ogbuachi, Anna Reale, Péter Suskovics, and Benedek Kovács. 2020. Context-Aware Kubernetes Scheduler for Edge-native Applications on 5G. *Journal of Communications Software and Systems* 16, 1 (2020), 85–94.
[15] Soojin Cho, Hongki Jo, Shinae Jang, Jongwoong Park, Hyung Jo Jung, Chung Bang Yun, Billie F. Spencer Jr, and Ju Won Seo. 2010. Structural health monitoring of a cable-stayed bridge using smart sensor technology: deployment and evaluation. *Smart Structures and Systems* 6, 5-6 (2010), 439–459.
[16] Byungkwon Choi, Jinwoo Park, Chunghan Lee, and Dongsu Han. 2021. pHPA: A proactive autoscaling framework for microservice chain. In *Proc. of APNet*. 65–71.
[17] Anirban Das, Stacy Patterson, and Mike Wittie. 2018. Edgebench: Benchmarking edge computing platforms. In *Proc. of IEEE/ACM UCC Companion*. 175–180.
[18] DeeptiDR. [n.d.]. K-means Demo. https://github.com/DeeptiDR/K-means_demo.
[19] Shuiguang Deng, Zhengzhe Xiang, Javid Taheri, Khoshkholghi Ali Mohammad, Jianwei Yin, Albert Zomaya, and Schahram Dustdar. 2020. Optimal application deployment in resource constrained distributed edges. *IEEE Transactions on Mobile Computing* (2020).
[20] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The Design and Operation of CloudLab.. In *Proc. of USENIX ATC*. 1–14.
[21] EdgeX Foundry. [n.d.]. EdgeX. https://www.edgexfoundry.org/.
[22] Ghofrane El Haj Ahmed, Felipe Gil-Castiñeira, and Enrique Costa-Montenegro. 2021. KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters. *Software: Practice and Experience* 51, 2 (2021), 213–234.
[23] Xenofon Foukas and Bozidar Radunovic. 2021. Concordia: teaching the 5G vRAN to share compute. In *Proc. of ACM SIGCOMM*. 580–596.
[24] Cloud Native Computing Foundation. [n.d.]. Volcano. https://github.com/volcano-sh/volcano.
[25] Gartner, Inc. 2020. Predicts 2021: Building on Cloud Computing as the New Normal. https://www.gartner.com/en/documents/3994453/.
[26] Gaoyang Guan, Wei Dong, Jiadong Zhang, Yi Gao, Tao Gu, and Jiajun Bu. 2019. Queec: QoE-aware edge computing for complex IoT event processing under dynamic workloads. In *Proc. of ACM TURC*. 1–5.

[27] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C Nachiappan, Mahmut Taylan Kandemir, and Chita R Das. 2020. Fifer: Tackling resource underutilization in the serverless era. In *Proc. of ACM/IFIP/USENIX Middleware.* 280–295.
[28] Yiwen Han, Shihao Shen, Xiaofei Wang, Shiqiang Wang, and C.M. Victor Leung. 2021. Tailored Learning-Based Scheduling for Kubernetes-Oriented Edge-Cloud System. In *Proc. of IEEE INFOCOM.*
[29] Tianshu Hao, Yunyou Huang, Xu Wen, Wanling Gao, Fan Zhang, Chen Zheng, Lei Wang, Hainan Ye, Kai Hwang, Zujie Ren, et al. 2019. Edge AIBench: towards comprehensive end-to-end edge computing benchmarking. In *Benchmarking, Measuring, and Optimizing: First BenchCouncil International Symposium.* Springer, 23–30.
[30] Shihong Hu, Weisong Shi, and Guanghui Li. 2022. CEC: A Containerized Edge Computing Framework for Dynamic Resource Provisioning. *IEEE Transactions on Mobile Computing* (2022).
[31] IBM. 2023. The true benefits of moving to containers. https://developer.ibm.com/articles/true-benefits-of-moving-to-containers-1/.
[32] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. 1984. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory* (1984).
[33] Deepak Janardhanan and Enda Barrett. 2017. CPU workload forecasting of machines in data centers using LSTM recurrent neural networks and ARIMA models. In *Proc. of IEEE ICITST.* 55–60.
[34] Kuljeet Kaur, Sahil Garg, Georges Kaddoum, Syed Hassan Ahmed, and Mohammed Atiquzzaman. 2019. KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem. *IEEE Internet of Things Journal* 7, 5 (2019), 4228–4237.
[35] James Kennedy and Russell Eberhart. 1995. PArticle swarm optimization. In *Proc. of IEEE ICNN.* 1942–1948.
[36] Hermann Kopetz and Wilfried Steiner. 2022. *Real-time systems: design principles for distributed embedded applications.* Springer Nature.
[37] kubernetes sigs. [n.d.]. kube-batch. https://github.com/kubernetes-sigs/kube-batch.
[38] Yongbo Li, Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. Mobiqor: Pushing the envelope of mobile edge computing via quality-of-result optimization. In *Proc. of IEEE MobiSys.* 1261–1270.
[39] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge assisted real-time object detection for mobile augmented reality. In *Proc. of ACM MobiCom.* 1–16.
[40] Dumitrel Loghin, Lavanya Ramapantulu, and Yong Meng Teo. 2019. Towards Analyzing the Performance of Hybrid Edge-Cloud Processing. In *Proc. of IEEE EDGE.* 87–94.
[41] Q. Luo, C. Li, T. H. Luan, and W. Shi. 2020. Collaborative Data Scheduling for Vehicular Edge Computing via Deep Reinforcement Learning. *IEEE Internet of Things Journal* 7, 10 (2020), 9637–9650. https://doi.org/10.1109/JIOT.2020.2983660
[42] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. 2018. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *Proc. of ACM/IEEE SEC.* 286–299.
[43] Víctor Medel, Unai Arronategui, José Ángel Bañares, Rafael Tolosana, and Omer Rana. 2019. Modeling, Characterising and Scheduling Applications in Kubernetes. In *Proc. of Springer GECON.* 291–294.
[44] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. 2016. Modelling performance & resource management in kubernetes. In *Proc. of ACM/IEEE UCC.* 257–262.
[45] Víctor Medel, Rafael Tolosana-Calasanz, José Ángel Bañares, Unai Arronategui, and Omer F Rana. 2018. Characterising resource management performance in Kubernetes. *Computers & Electrical Engineering* 68 (2018), 286–297.
[46] Jiaying Meng, Haisheng Tan, Chao Xu, Wanli Cao, Liuyan Liu, and Bojie Li. 2019. Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In *Proc. of IEEE INFOCOM.* 2287–2295.
[47] Matthias Meyer, Timo Farei-Campagna, Akos Pasztor, Reto Da Forno, Tonio Gsell, Jérome Faillettaz, Andreas Vieli, Samuel Weber, Jan Beutel, and Lothar Thiele. 2019. Event-triggered natural hazard monitoring with convolutional neural networks on the edge. In *Proc. of ACM/IEEE IPSN.* 73–84.
[48] Mariyam Mirza, Joel Sommers, Paul Barford, and Xiaojin Zhu. 2007. A machine learning approach to TCP throughput prediction. *ACM SIGMETRICS Performance Evaluation Review* 35, 1 (2007), 97–108.
[49] NetApp. 2023. What are containers? https://www.netapp.com/devops-solutions/what-are-containers/.
[50] Cao Ngoc Nguyen, Jik-Soo Kim, and Soonwook Hwang. 2016. KOHA: Building a Kafka-Based Distributed Queue System on the Fly in a Hadoop Cluster. In *Proc. of IEEE FAS*W.* 48–53.
[51] openalpr. [n.d.]. openalpr. https://github.com/openalpr/openalpr.
[52] Maria Rodriguez and Rajkumar Buyya. 2020. Container Orchestration With Cost-Efficient Autoscaling in Cloud Computing Environments. In *Handbook of Research on Multimedia Cyber Security.* IGI Global, 190–213.
[53] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proc. of IEEE CLOUD.* 500–507.
[54] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at Google. In *Proc. of ACM EuroSys.* 1–16.
[55] Jose Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. 2019. Towards network-aware resource provisioning in kubernetes for fog computing applications. In *Proc. of IEEE NetSoft.* 351–359.

[56] Royal Sequiera and Jimmy Lin. 2017. Finally, a downloadable test collection of tweets. In *Proc. of ACM SIGIR*. 1225–1228.
[57] Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. 2016. Beam: Ending monolithic applications for connected devices. In *Proc. of USENIX ATC*. 143–157.
[58] Smruthi Sridhar and Matthew E Tolentino. 2017. Evaluating voice interaction pipelines at the edge. In *Proc. of IEEE EDGE*. 248–251.
[59] Rainer Storn and Kenneth Price. 1997. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization* 11, 4 (1997), 341–359.
[60] Tejun Heo. 2023. Control Group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.
[61] Laszlo Toka, Gergely Dobreff, Balazs Fodor, and Balazs Sonkoly. 2020. Adaptive AI-based auto-scaling for Kubernetes. In *Proc. of IEEE/ACM CCGRID*. 599–608.
[62] Paul Townend, Stephen Clement, Dan Burdett, Renyu Yang, Joe Shaw, Brad Slater, and Jie Xu. 2019. Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes. In *Proc. of IEEE SOSE*. 156–15610.
[63] Oana-Mihaela Ungureanu, Călin Vlădeanu, and Robert Kooij. 2019. Kubernetes cluster optimization using hybrid shared-state scheduling framework. In *Proc. of ACM ICFNDS*. 1–12.
[64] Peter JM Van Laarhoven and Emile HL Aarts. 1987. Simulated annealing. In *Simulated annealing: Theory and applications*. Springer, 7–15.
[65] Blesson Varghese, Nan Wang, David Bermbach, Cheol-Ho Hong, Eyal De Lara, Weisong Shi, and Christopher Stewart. 2021. A survey on edge performance benchmarking. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–33.
[66] José Luis Vázquez-Poletti and Ignacio Martín Llorente. 2018. Serverless Computing: From Planet Mars to the Cloud. *Computing in Science Engineering* 20, 6 (2018), 73–79.
[67] Lei Wang, Jin Pan, and Li-cheng Jiao. 2000. The immune algorithm. *Acta Electronica Sinica* 28, 7 (2000), 74–78.
[68] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. 2020. Metis: Learning to Schedule Long-Running Applications in Shared Container Clusters at Scale. In *Proc. of ACM/IEEE SC*. 1–17.
[69] Yifan Wang, Shaoshan Liu, Xiaopei Wu, and Weisong Shi. 2018. CAVBench: A benchmark suite for connected and autonomous vehicles. In *Proc. of IEEE/ACM SEC*. 30–42.
[70] Zhang Wei-guo, Ma Xi-lin, and Zhang Jin-zhong. 2018. Research on Kubernetes' Resource Scheduling Scheme. In *Proc. of ICCNS*. 144–148.
[71] Zhaojie Wen, Yishuo Wang, and Fangming Liu. 2022. StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows. In *Proc. of IEEE INFOCOM*. 1868–1877.
[72] Thomas Weng, Anthony Nwokafor, and Yuvraj Agarwal. 2013. BuildingDepot 2.0: An Integrated Management System for Building Analysis and Control. In *Proc. of ACM BuildSys*.
[73] Darrell Whitley. 1994. A genetic algorithm tutorial. *Statistics and computing* 4, 2 (1994), 65–85.
[74] Lukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. 2021. NetMARKS- Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh. In *Proc. of IEEE INFOCOM*.
[75] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. 2020. Approximate query service on autonomous iot cameras. In *Proc. of ACM MobiSys*. 191–205.
[76] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic scaling for microservices with an online learning approach. In *Proc. of IEEE ICWS*. 68–75.
[77] Chaoqun Yue, Ruofan Jin, Kyoungwon Suh, Yanyuan Qin, Bing Wang, and Wei Wei. 2017. LinkForecast: cellular link bandwidth prediction in LTE networks. *IEEE Transactions on Mobile Computing* 17, 7 (2017), 1582–1594.
[78] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *Proc. of USNIX ATC*. 1049–1062.
[79] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proc. of ACM ASPLOS*. 167–181.
[80] Anqi Zhao, Qiang Huang, Yiting Huang, Lin Zou, Zhengxi Chen, and Jianghang Song. 2019. Research on resource prediction model based on kubernetes container auto-scaling technology. In *IOP Conference Series: Materials Science and Engineering*, Vol. 569. 052092.
[81] Zhiheng Zhong and Rajkumar Buyya. 2020. A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources. *ACM Transactions on Internet Technology (TOIT)* 20, 2 (2020), 1–24.