Tango: Harmonious Management and Scheduling for Mixed Services Co-located among Distributed Edge-Clouds

Yicheng Feng
Tianjin University
Tianjin, China
yichengfeng@tju.edu.cn

Shihao Shen
Tianjin University
Tianjin, China
shenshihao@tju.edu.cn

Mengwei Xu
Beijing University of Posts and Telecommunications
Beijing, China
mwx@bupt.edu.cn

Yuanming Ren
Tianjin University
Tianjin, China
renyuanming@tju.edu.cn

Xiaofei Wang*
Tianjin University
Tianjin, China
xiaofeiwang@tju.edu.cn

Victor C.M. Leung
Shenzhen University
Shenzhen, China
vleung@ieee.org

Wenyu Wang
Paiou Cloud Computing Co., Ltd.
Shanghai, China
wayne@pplabs.org

ABSTRACT

Co-locating Latency-Critical (LC) and Best-Effort (BE) services in edge-clouds is expected to enhance resource utilization. However, this mixed deployment encounters unique challenges. Edge-clouds are heterogeneous, distributed, and resource-constrained, leading to intense competition for edge resources, making it challenging to balance fluctuating co-located workloads. Previous works in cloud datacenters are no longer applicable since they do not consider the unique nature of edges. Although very few works explicitly provide specific schemes for edge workload co-location, these solutions fail to address the major challenges simultaneously.

In this paper, we propose *Tango*, a harmonious management and scheduling framework for *Kubernetes*-based edge-cloud systems with mixed services, to address these challenges. *Tango* incorporates novel components and mechanisms for elastic resource allocation and two traffic scheduling algorithms that effectively manage distributed edge resources. *Tango* demonstrates harmony not only in the compatible mixed services it supports, but also in the collaborative solutions that complement each other. Based on a backwards compatible design for *Kubernetes*, *Tango* enhances *Kubernetes* with automatic scaling and traffic scheduling capabilities. Experiments on large-scale hybrid edge-clouds, driven by real workload traces, show that *Tango* improves the system resource utilization by 36.9%, QoS-guarantee satisfaction rate by 11.3%, and throughput by 47.6%, compared to state-of-the-art approaches.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Real-time system architecture**; • **Networks → Cloud computing**.

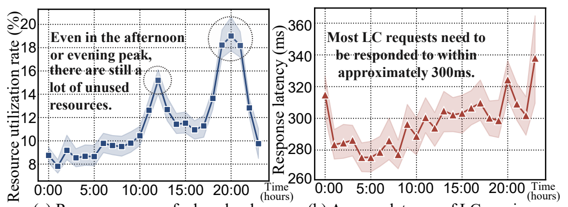
KEYWORDS

mixed service, resource management and scheduling, edge-clouds

ACM Reference Format:
Yicheng Feng, Shihao Shen, Mengwei Xu, Yuanming Ren, Xiaofei Wang*, Victor C.M. Leung, and Wenyu Wang. 2023. Tango: Harmonious Management and Scheduling for Mixed Services Co-located among Distributed Edge-Clouds. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605589

1 INTRODUCTION

Edge computing inherits the scalability of cloud computing and has enabled the distribution of computing resources in massive, small clusters, which provides more agile and in-proximity services to end-users [30, 32, 38]. As a complementary solution to cloud computing, edge computing is well-suited to support Latency-Critical (LC) services [24], including virtual reality, smart factories, and self-driving cars, due to its lower latency and ability to provide better quality-of-service (QoS).



(a) Resource usage of edge clouds. (b) Average latency of LC services.

Figure 1: Measurement of industrial edge-clouds.

Individually hosting LC services in the edge introduces severe resource underutilization [29]. This is due to the high variability of edge workloads over time, leading developers to over-provision resources to handle peak loads. Our measurements on representative edge-clouds in the wild (Figure 1(a)) [1] indicate that the average utilization of edge-cloud resources is below 20%. This presents significant potential for improvements. One promising solution to address the resource underutilization is to co-locate Best-Effort (BE) services, such as data analytics and AI model training, with LC services on the same edge-cloud, which has been widely adopted in cloud datacenters, such as Google's Borg [36]. This approach can help exploit any unused resources by LC services [23, 26].

[1]PPIO Edge Cloud (www.ppio.cn) supports for the production dataset.

ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0843-5/23/08…$15.00
https://doi.org/10.1145/3605573.3605589

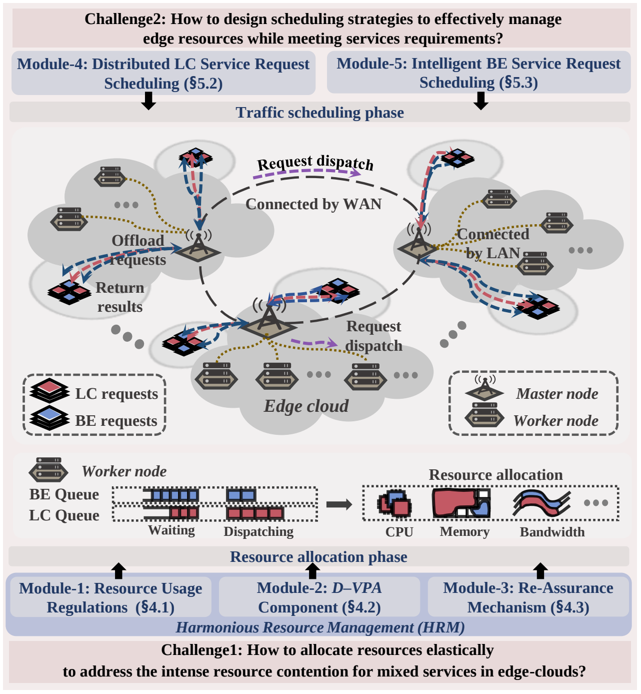
Yet, co-locating LC and BE services on edges poses two unique challenges. Firstly, unlike cloud datacenters with relatively abundant resources, edge-clouds are usually heterogeneous and resource-constrained, making competition for resources more intense [33]. Given the strict QoS targets of LC services, such as latency, as illustrated in Figure 1(b)[1]note1, it is challenging to address severe resource contention using elastic resource allocation approaches that are applicable to edge-clouds [22, 40]. Secondly, edge resources are distributed and scattered, while user requests' loads are uneven and fluctuating across geographical locations [24, 39]. This indicates that supply-demand balance needs to be achieved through workload scheduling [30, 37]. Collaborative management of decentralized resources is the key to edge-clouds [24, 41]. If scattered resources cannot be efficiently utilized by computation offloading (i.e., traffic scheduling), it will result in load imbalance and poor QoS. Therefore, the design of computation offloading schemes that effectively manage edge resources while meeting service requirements is challenging for edge workload co-location.

While cloud datacenters face similar resource allocation and traffic scheduling challenges [5, 14, 27, 36], the nature of edges has made these challenges much more complex, rendering traditional cloud datacenter solutions inapplicable. Firstly, resource-constrained edge-clouds require aggressive repurposing and fast provisioning [13], which makes elasticity solutions that rely on a pool of "hot" workers overly rigid and expensive [12, 25]. Secondly, simple static traffic policies used in cloud datacenters are unable to efficiently manage scattered edge resources [15, 28]. For instance, LC service requests may be assigned to an overloaded edge-cloud, leading to poor QoS and load imbalance, even if other nearby edge-clouds are relatively idle. Despite a few works providing specific schemes for edge workload co-location [23, 26], these solutions fail to address both major challenges simultaneously.

In this paper, we propose *Tango*, the first collaborative solution that optimizes resource allocation and computation offloading for edge co-location, aiming to improve resource utilization, throughput, and QoS for LC services in edge-cloud systems.

Figure 2 illustrates the design of *Tango*, which addresses two major challenges in edge-cloud systems using five modules compatible with *Kubernetes (K8s)* [6]. For flexible resource allocation, *Tango* introduces the ***Harmonious Resource Management (HRM)*** approach. It includes three modules: (1) *resource usage regulation* to prioritize services during processing and offloading, (2) *dynamic vertical pod autoscaler (D-VPA)* for rapid resource provisioning, and (3) a *QoS re-assurance mechanism* for dynamic resource adjustment. To efficiently manage decentralized edge resources, *Tango* employs ***two traffic scheduling algorithms***: (1) a *distributed real-time decision-making algorithm for network flows* for LC service request dispatching and (2) a *centralized intelligent learning algorithm based on Graph Neural Network (GNN)* for BE service request scheduling with adaptive adjustment capability.

Tango exhibits harmony in two respects. First, *Tango* supports mixed services, ensuring that the QoS of LC services is given priority while the BE services are dynamically adjusted based on the varying load of LC services. Second, the collaborative solutions offered by *Tango* to address the challenges work together in harmony. Specifically, *Tango's HRM* defines the resource priority using two traffic scheduling algorithms, which ensures local scheduling



Challenge2: How to design scheduling strategies to effectively manage edge resources while meeting services requirements?
Module-4: Distributed LC Service Request Scheduling (§5.2)
Module-5: Intelligent BE Service Request Scheduling (§5.3)
Traffic scheduling phase
Request dispatch
Connected by WAN
Offload requests
Return results
Connected by LAN
Request dispatch
LC requests
BE requests
Edge cloud
Master node
Worker node
Worker node
BE Queue
LC Queue
Waiting
Dispatching
Resource allocation
CPU
Memory
Bandwidth
Resource allocation phase
Module-1: Resource Usage Regulations (§4.1)
Module-2: D–VPA Component (§4.2)
Module-3: Re-Assurance Mechanism (§4.3)
Harmonious Resource Management (HRM)
Challenge1: How to allocate resources elastically to address the intense resource contention for mixed services in edge-clouds?

Figure 2: An overview of *Tango*.

efficiency on each edge-cloud. Meanwhile, *Tango*'s two scheduling policies optimize edge-clouds overall according to edge-cloud resources maintained by *HRM*.

The main contributions of this paper are summarized as follows:

- We introduce *Tango*, a framework for harmonious management and scheduling of edge workloads. *Tango* is designed to improve system resource utilization and throughput while ensuring the QoS of LC services by using a series of mechanisms and components that are compatible with *K8s* for edge-clouds.
- We present *HRM*, a comprehensive solution designed for flexible resource allocation in the edge. *HRM* includes resource usage regulations, the *D-VPA* component, and the QoS re-assurance mechanism. *HRM* supports resource segregation and targeted allocation by adjusting resource provisioning elastically.
- We develop two scheduling algorithms for edge workload co-location that efficiently manage decentralized edge resources: a distributed network flow algorithm for LC service requests, and a centralized intelligent learning algorithm based on GNN for BE service requests.
- We conduct experiments on large-scale hybrid edge-clouds driven by real workload traces, and compare our approach to the state-of-the-art. The results show that *Tango* can improve system resource utilization by 36.9% and QoS-guarantee satisfaction rate by 14.1%, while further improving the throughput by 58.9%.

Open-source. *Tango* is available as an open-source project at https://github.com/fwyc0573/Tango .

2 BACKGROUND AND RELATED WORK

2.1 Container Orchestration Platforms

Many publicly-available container orchestration platforms (COPs), such as *Borg* [36], *K8s* [6], *Mesos* [21], and *Docker Swarm* [10] offer comparable functionality for diverse workloads on large clusters, but are not sufficiently suitable for edge workload co-location. For instance, *K8s* relies on autoscaling and traffic scheduling mechanisms which pose challenges for mixed services in edge-clouds. Horizontal scaling [3], which adjusts the number of instances as part of autoscaling, is relatively time-consuming for millisecond-level LC services due to long container start-up time. While *K8s*'s vertical scaling component [11] can modify the configuration of instance resources, it causes downtime since it relies on a delete-and-rebuild approach. In terms of traffic scheduling, *K8s* only provides simplistic policies such as round-robin [9], which fail to consider service priority or the heterogeneous nature of the edge.

2.2 Resource Allocation

There have been several works that focus on resource allocation in co-location scenarios, including *PROMPT* [29], *Borg* [36], *Edgeiso* [26], *Koordinator* [5], and *Twig* [27]. These works use a combination of hardware and software mechanisms to isolate and allocate resources, providing some relief from the resource allocation challenge. However, these works lack attention to the unique nature of edge resources, and do not effectively address the challenge of organizing scattered edge resources through traffic scheduling strategies. In addition, resource reservation is a typical solution to address fluctuating traffic loads and achieve resource elasticity in cloud datacenters [12, 43]. For example, Ambati *et al.* [12] proposed a framework that maintains an extra pool of transient virtual machines for requests. However, these solutions are too expensive for resource-constrained edges and have a large time overhead for resource provisioning, which can be fatal for LC service requests.

2.3 Computation Offloading

Many of the traffic scheduling policies used in cloud datacenters today, such as *Maglev* [15] and *Ananta* [28], adopt simple static strategies, which may not be suitable for edge co-location scenarios. These policies fail to address service prioritization and the heterogeneity of edge-clouds. While some existing work on edge-clouds involves well-designed computation offloading policies, they only optimize QoS for LC services or throughput for BE services and cannot address the co-location scenario [19, 22, 31, 35]. Moreover, many of these works are not designed to integrate with COPs or other existing cloud infrastructures. Although some edge co-location works include computation offloading strategies [23], the challenge of resource allocation has not been adequately addressed.

3 TANGO OVERVIEW

We present the architecture design of *Tango* in Figure 3. *Tango* is designed with backward compatibility in mind and extends *K8s* with automatic scaling and traffic scheduling capabilities that are specifically optimized for edge-clouds. The main components of *Tango* are as follows:

- ***LC traffic dispatcher***. *Tango* runs a *LC traffic dispatcher* ❶ on each master node to dispatch LC service requests from users. Using a custom distributed scheduling algorithm (§5.2), the dispatcher makes real-time decisions based on information from the *state storage* ❷. The state storage not only stores the status of nearby edge-clouds but also periodically receives metrics, such as resource usage, round-trip time, and the QoS, which are pushed by Prometheus [8] and the *QoS detector* ❹, respectively.

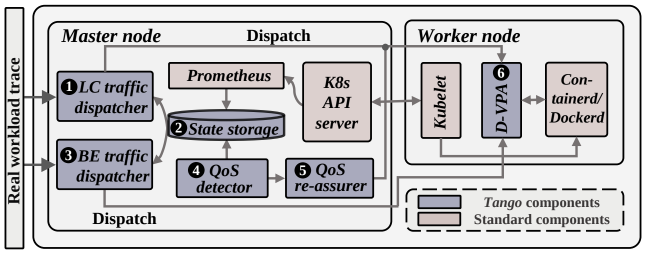


Figure 3: *Tango* architecture.

- ***BE traffic dispatcher***. To provide centralized scheduling of BE service requests (§5.3), a *BE traffic dispatcher* ❸ is deployed on the master node of the central edge-cloud cluster. [2] While the *LC traffic dispatcher* prioritizes request QoS, the *BE traffic dispatcher* receives BE service requests forwarded from other edge-clouds and optimizes system throughput through centralized decisions at the cost of transmission latency. Both LC and BE traffic dispatchers follow resource usage regulations (§4.1) to balance mixed workloads.
- ***QoS re-assurer***. Each master node deploys a *QoS re-assurer* ❺, which receives QoS of LC requests from the *QoS detector* ❹ and calculates the amount of resource adjustment based on the status in the past period, encapsulating it in the packet of scheduled requests, in line with the *QoS re-assurance mechanism* (§4.3).
- ***D-VPA component***. The *D-VPA* component ❻ (§4.2) enables elastic resource allocation without the need for a delete-and-rebuild approach used in the current *K8s-VPA* plugin [11]. It achieves this by dynamically controlling resource limits of pods and containers, facilitating vertical scaling without disruptions.

Operation. Next, we provide a high-level description of the operation of *Tango* and how it aligns with the three-step **dispatch–allocate–adjust** process: (1) Upon receiving service requests, the master node places them into either the LC or BE scheduling queues. LC service requests are promptly **dispatched** to target nodes by the *LC traffic dispatcher* of each cluster. In contrast, BE service requests are uniformly forwarded to the central cluster for centralized **dispatching** by its *BE traffic dispatcher*. (2) Once dispatched, the cluster forwards requests to target nodes. On the worker node, the *D-VPA* component **allocates** resources (minimum requirements) to containers by dynamically modifying the resource limits corresponding to the service request type. After processing the requests, the allocated resources are reclaimed. (3) The *QoS detector* **adjusts** the minimum amount of resources requested from the *D-VPA* component based on the QoS of LC service requests. Meanwhile, the *D-VPA* component dynamically **adjusts** the resource allocation of containers, following the regulations of resource usage.

4 DESIGN OF HRM

HRM is an elastic resource management solution that targets edge-clouds. It consists of the *D-VPA* component (§4.2) responsible for flexible resource allocation at a low level. It also includes regulations

[2] In our scenario, *Tango* selects an edge-cloud cluster that is (i) geographically central and (ii) more resource-rich for centralized scheduling of BE service requests.

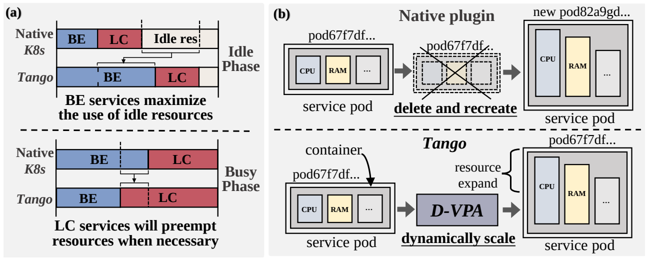


Figure 4: Resource usage regulations (a) and *D-VPA* (b).

of resource usage (§4.1) and the QoS re-assurance mechanism (§4.3) that guide and adjust *D-VPA*'s resource allocation.

4.1 Resource Usage Regulations

To manage resource allocation for mixed services effectively throughout the scheduling and processing phases, *Tango* implements specific regulations. In conjunction with the QoS level mechanism of *K8s*, LC services are assigned a higher priority than BE services.

The resources available for scheduling and processing LC service requests include both idle resources and resources currently being used by BE services, with preference given to the former. As shown in Figure 4(a), the BE services aim to maximize idle resources to speed up request processing and improve throughput. If the current idle resources in edge-clouds are insufficient to meet the minimum requirements of pending LC service requests, preemption is allowed. LC service containers receive shares directly from BE services for compressible resources, such as CPU and bandwidth. However, incompressible resources, such as memory and disk, are freed up by evicting and restarting running BE services at a later time.

4.2 *D-VPA* Component

The nature of edge-clouds being resource-constrained makes it important to efficiently utilize resources and quickly allocate them. Therefore, expensive solutions such as pre-reserving resources (by setting high resource limits in advance) or solutions that have high start-up latency such as real-time instance creation are likely not suitable. [3] Previous research [40] has shown a significant difference between actual resource utilization and the utilization presented by the cluster manager, resulting in resource wastage.

Pain Points. The resource allocation settings of native *K8s* are only involved in initialization, meaning that the *K8s* resource list (such as CPU or memory) cannot be modified while containers are running. Even the *K8s-VPA* [11] plugin, specifically designed for vertical expansion and contraction, uses a delete and rebuild approach that interrupts the running container. This may be due to the fact that *K8s* currently does not support runtime modification of resources for pods (i.e., the smallest unit of *K8s*) and containers.

K8s CGroup Control. As shown in Figure 4(b), *Tango's D-VPA* provides a dynamic resource scaling solution through the addition of an extra control flow to K8s Control Group (CGroup), instead of using the disruptive delete-and-rebuild method. The modification process involves the pod-level CGroups in addition to the

[3] *Tango*'s scenario aligns with the edge-cloud enterprise discussed in §1, where fixed types of containerized applications, such as cloud rendering, audio, and video, run continuously on the edge-clouds to serve users with specific requirements. The actual resource utilization of containers is influenced by fluctuating workloads.

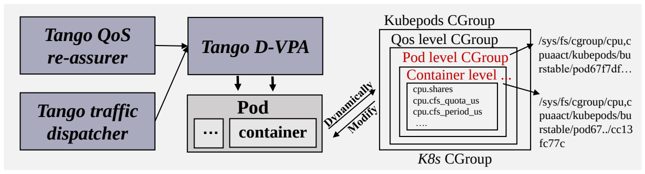


Figure 5: *D-VPA*'s control flow to K8s CGroups.

container-level CGroups (Figure 5), and the modifications must be sequential to prevent failure. Specifically, for expansion cases, the pod-level CGroups are modified first, followed by the container-level CGroups. For shrinking cases, the order is reversed. *D-VPA* allocates the minimum resources required for each service request to be processed and reclaims them upon completion. The resource allocation is dynamically adjusted based on resource usage regulations and the QoS assurance mechanism.

4.3 QoS Re-Assurance Mechanism

The volatility of system load has a significant impact on the latency of LC service requests. To address this issue, *Tango* implements a re-assurance mechanism that can mitigate fluctuations in the QoS. The processing latency of LC service requests on each worker node is collected within a time window of 100ms. We introduce the slack score $\delta^k(n_b^i)$, defined as $1 - \xi_i^k/\gamma^k$, where ξ_i^k represents the tail latency (95^{th} percentile) of service k within a time window at node n_b^i. The QoS target of service k is denoted by γ^k.

A negative slack score indicates that the request latency fails to meet the QoS target, and the severity of the violation increases as the slack score decreases. Based on the slack score concept, we empirically establish two thresholds (α and β) and classify three levels of quality performance (excellent, stable, and poor). Algorithm 1 depicts the resource adjustment strategies for different performance levels. Note that to minimize resource perturbations, the mechanism operates at a high frequency with a small proportion, ensuring that resource adjustments are timely and smooth.

5 DESIGN OF SCHEDULING ALGORITHMS

In this section, we formally present the system model of our edge-cloud scenario and describe the optimization objective. We then introduce two traffic scheduling algorithms, one tailored for LC service requests and the other for BE service requests.

5.1 Problem Formulation

5.1.1 *Edge-Cloud System.*

An *edge-cloud system* typically consists of multiple *edge-cloud clusters*. Each cluster comprises numerous *edge-clouds* that can

Algorithm 1: The re-assurance mechanism in *HRM*

```
1 for each worker node do in parallel
2     for each service k do in parallel
3         if $\delta^k(n_b^i) < \alpha$ then
4             Increase the minimum requested resource amount for
              service $k$ on node $n_b^i$;
5         else if $\delta^k(n_b^i) > \beta$ then
6             Decrease the minimum requested resource amount for
              service $k$ on node $n_b^i$;
```

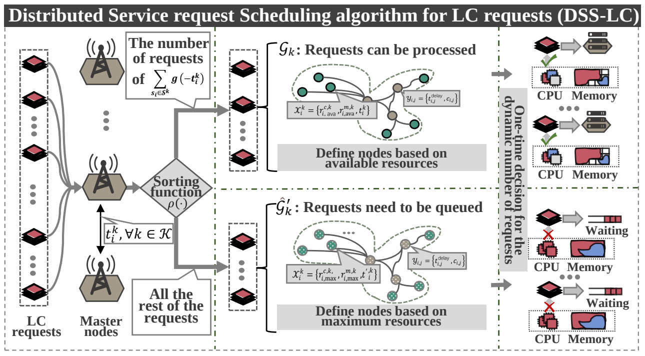

Distributed Service request Scheduling algorithm for LC requests (DSS-LC)
The number of requests of $\sum_{s_i \in \mathcal{S}^k} g(-t_i^k)$
$\mathcal{G}_k$: Requests can be processed
$\mathcal{X}_i^k = \{r_{i,\text{ava}}^{c,k}, r_{i,\text{ava}}^{m,k}, t_i^k\}$
$\mathcal{Y}_{i,j} = \{t_{i,j}^{\text{delay}}, c_{i,j}\}$
Define nodes based on available resources
Sorting function $\rho(\cdot)$
$t_i^k, \forall k \in \mathcal{K}$
$\hat{\mathcal{G}}_k'$: Requests need to be queued
$\mathcal{X}_i^k = \{r_{i,\max}^{c,k}, r_{i,\max}^{m,k}, t'^k_i\}$
$\mathcal{Y}_{i,j} = \{t_{i,j}^{\text{delay}}, c_{i,j}\}$
Define nodes based on maximum resources
All the rest of the requests
LC requests
Master nodes
One-time decision for the dynamic number of requests
CPU
Memory
Waiting

Figure 6: Distributed LC service request scheduling.

function as either master nodes or worker nodes. Here, we provide a detailed explanation of the concepts involved:

- **Edge-Cloud cluster.** We define the set of edge-cloud clusters as $\mathcal{B}$, where $b \in \mathcal{B}$ represents a single cluster. An edge-cloud cluster b consists of m_b edge-clouds. These edge-clouds are connected through a Local Area Network (LAN), and the clusters themselves are connected via a Wide Area Network (WAN).
- **Master nodes.** Incoming requests are received by the master node, which acts as an edge access point (eAP). In addition, the master node serves as a controller and decision maker within the edge-cloud cluster, with at least one present in number.
- **Worker nodes.** The worker node executes container instances. It processes service requests forwarded by the master node and returns results. Typically, an edge-cloud cluster comprises multiple worker nodes.

5.1.2 *System Optimization Objective*.

The optimization objective of the system is determined by two factors: (i) the **QoS-guarantee satisfaction rate**, which is the percentage of completed LC service requests that meet the QoS targets (tail latency being the QoS metric); and (ii) the **long-term throughput**, which is the total number of completed BE service requests over time.

Concretely, we define the optimization objective U for the edge-cloud system as maximizing *the rate of QoS-guarantee satisfaction* $\phi = \sum_{t=0}^{+\infty} \sum_{b \in \mathcal{B}} q_{b,t} / \sum_{t=0}^{+\infty} \sum_{b \in \mathcal{B}} Q_{b,t}$ for LC service requests while improving *the long-term throughput* $\phi' = \sum_{t=0}^{+\infty} \sum_{b \in \mathcal{B}} q'_{b,t}$ for BE service requests as much as possible. The number of LC service requests arriving at edge-cloud cluster b at time t is denoted by $Q_{b,t}$, and $q_{b,t}$ is the number of completed request in $Q_{b,t}$ that meet the QoS targets. Similarly, the number of completed BE service requests at time t in the edge-cloud cluster b is denoted by $q'_{b,t}$. The optimization objective U of *Tango* can be expressed as:

$$\max_{\{\hat{\pi}_{b,t}: b \in \mathcal{B}\}, \tilde{\pi}_t} U = \max \sum_{t=0}^{+\infty} \sum_{b \in \mathcal{B}} q'_{b,t} \max \frac{\sum_{t=0}^{+\infty} \sum_{b \in \mathcal{B}} q_{b,t}}{\sum_{t=0}^{+\infty} \sum_{b \in \mathcal{B}} Q_{b,t}}, \tag{1}$$

where $\{\hat{\pi}_{b,t} : b \in \mathcal{B}\}$ denotes the LC-oriented policy deployed on edge-cloud cluster b and $\tilde{\pi}_t$ denotes the BE-oriented policy deployed on the central edge-cloud cluster.

5.2 Request Scheduling of LC Services

One key intuition in scheduling LC service requests is to use a distributed approach, where quick scheduling decisions are made once requests are received by an eAP. Previous research [20] shows that distributed scheduling significantly reduces time overhead in edge clouds, which is crucial for LC service requests. Our edge-cloud production dataset reveals a round-trip time from the edge-cloud to the central cluster can exceed 97ms, which is very detrimental to the QoS (close to 30% of the average QoS targets). This is also one of the reasons why we choose not to use centralized scheduling for LC requests. Consequently, we formulate the scheduling problem as a Multi-Commodity Network Flow (MCNF) problem, and implement the **Distributed Service request Scheduling algorithm for LC service requests (*DSS-LC*)**, as shown in Figure 6 and Alg. 2.

5.2.1 *Multi-commodity Network Flow Problem*.

The *DSS-LC* simultaneously creates a graph $\mathcal{G}_k$ for each LC request type $k \in \mathcal{K}$. Specifically, for a given request type $k \in \mathcal{K}$, the corresponding graph $\mathcal{G}_k = (\mathcal{S}^k, \zeta^k)$ contains systematic information about the request type. Here, $\mathcal{S}^k$ represents the set of nodes and ζ^k represents the set of edges in the graph. [4] For any edge $(s_i, s_j) \in \zeta^k$, both s_i and s_j belong to $\mathcal{S}^k$. The mathematical formulation of the MCNF problem is as follows.

- **The set of nodes $\mathcal{S}^k$.** For each edge node $s_i \in \mathcal{S}^k$, a set of node attributes $\mathcal{X}_i^k = \{r_{i,\text{total}}^{c,k}, r_{i,\text{ava}}^{c,k}, r_{i,\text{total}}^{m,k}, r_{i,\text{ava}}^{m,k}, t_i^k\}$ is defined to provide systematic information about request type k. [5] The available CPU and total CPU (i.e., the sum of both available and occupied CPU) of node s_i are denoted by $r_{i,\text{ava}}^{c,k}$ and $r_{i,\text{total}}^{c,k}$, respectively. Similarly, the available memory and total memory are represented by $r_{i,\text{ava}}^{m,k}$ and $r_{i,\text{total}}^{m,k}$, respectively. The supply and demand relationship of node s_i with respect to request type k is indicated by t_i^k. Specifically, when s_i is a master node, it means that the node has t_i^k pending requests (waiting in the scheduling queue). On the other hand, when s_i is a worker node, t_i^k denotes the node's capacity to host $|t_i^k|$ requests (see Eq. 2). Here, $r_i^{c,k}$ and $r_i^{m,k}$ are the minimum required CPU and memory for request type k, respectively. The values of $r_i^{c,k}$ and $r_i^{m,k}$ are dynamically adjusted by the QoS re-assurance mechanism described in §4.3.

$$t_i^k = -\min(r_{i,\text{ava}}^{c,\,k} / r_i^{c,k}, r_{i,\text{ava}}^{m,\,k} / r_i^{m,k}). \tag{2}$$

- **The set of edges ζ^k.** For each edge $(s_i, s_j) \in \zeta^k$, a set of edge attributes $\mathcal{Y}_{i,j}$ is defined as $\mathcal{Y}_{i,j} = \{t_{i,j}^{\text{delay}}, c_{i,j}\}$. Here, $t_{i,j}^{\text{delay}}$ represents the transmission delay between edge nodes s_i and s_j, while $c_{i,j}$ represents the transmission capacity between them.

5.2.2 *Algorithm Design of DSS-LC*.

The aim of the *DSS-LC* is to maximize the number of LC service requests transmitted while minimizing the overall transmission latency. Mathematically, we define the set of transmitted requests as $\mathcal{F}$. If $b_{i,j}^f$=1, request f is transmitted through (s_i, s_j); otherwise, $b_{i,j}^f$=0. Furthermore, we define $\mathcal{F}_{i,j}$ as the set of all requests sent from s_i to s_j, f as a request transmission stream, and γ^f as the

[4] Due to latency considerations, LC service requests can be dispatched to a local or geo-nearby clusters for processing (within 500km of local in our production dataset).
[5] The resources available for scheduling LC service requests are regulated in §4.1.

Algorithm 2: Distributed LC service request scheduling (DSS-LC)

```
1  for each master node do in parallel
2      Place received LC requests into the LC scheduling queue;
3      if the LC scheduling queue is not empty then
4          for each LC request type $k \in \mathcal{K}$ do in parallel
5              if $\sum_{s_i \in \mathcal{S}^k} g(t_i^k) \leq \sum_{s_i \in \mathcal{S}^k} g(-t_i^k)$ then
6                  Get node attributes $\mathcal{X}_i^k$ and edge attributes $\mathcal{Y}_{i,j}$ to
                     construct the graph $\mathcal{G}_k$;
7                  Use the ortool to get the scheduling routing path;
8              if $\sum_{s_i \in \mathcal{S}^k} g(t_i^k) > \sum_{s_i \in \mathcal{S}^k} g(-t_i^k)$ then
9                  Requests are divided into $\mathcal{R}_k$ and $\mathcal{R}_k'$;
10                 Get node attributes $\mathcal{X}_i^k$ and edge attributes $\mathcal{Y}_{i,j}$;
11                 Construct graphs $\mathcal{G}_k$ and $\hat{\mathcal{G}}_k'$, respectively;
12                 Use the ortool to get scheduling routing path;
13             Forward requests to target nodes according to the
                 scheduling routing path;
```

resource requirements for f. Thus, the optimization goal of *DSS-LC* can be expressed as follows:

$$\min_{(s_i,s_j)} \max_{q_{b,t}} \sum_{f \in \mathcal{F}} \sum_{(s_i,s_j) \in E} b_{i,j}^f t_{i,j}^{\text{delay}} \tag{3}$$

$$\text{s.t.} \qquad \sum_{f \in \mathcal{F}_{i,j}} \gamma^f \leq c_{i,j}, \quad \forall (s_i,s_j) \in \zeta^k, \tag{4}$$

$$\begin{aligned} &\sum f \in \mathcal{F}_{m,j} - \sum f' \in \mathcal{F}_{j,n} \leq |t_j^k|, \quad \forall (s_j) \in \mathcal{S}^k, \\ &\exists (s_m, s_j) \in \zeta^k, \quad \exists (s_j, s_n) \in \zeta^k, \quad \text{if} \quad \mathrm{t}_\mathrm{j}^\mathrm{k} \leq 0, \end{aligned} \tag{5}$$

$$\begin{aligned} &\sum f' \in \mathcal{F}_{j,n} - \sum f \in \mathcal{F}_{m,j} \leq t_j^k, \quad \forall (s_j) \in \mathcal{S}^k, \\ &\exists (s_m, s_j) \in \zeta^k, \quad \exists (s_j, s_n) \in \zeta^k, \quad \text{if} \quad \mathrm{t}_\mathrm{j}^\mathrm{k} > 0, \end{aligned} \tag{6}$$

where Eq.4 ensures that the sum of requests transmitted cannot exceed the upper limit, Eq.5 ensures that the number of requests received by each node cannot exceed its processing capacity, and Eq.6 means that the number of requests sent by each node is expected to be no more than the sum of initially owned and the number of received. We focus on the fundamental variables and constraints to maintain brevity. However, in practice, different traffic engineering system requirements and routing protocols (e.g., MPLS tunneling selection, OSPF) can be incorporated into the problem formulation.

The *DSS-LC* algorithm handles requests of each type $k \in \mathcal{K}$ in two cases. The first case arises when $\sum_{s_i \in \mathcal{S}^k} t_i^k \leq 0$, which means that the number of pending requests is no more than the number that can be processed, i.e., the target scheduling node has the capacity to execute these requests. In this case, *DSS-LC* constructs the graph $\mathcal{G}_k$ as described in §5.2.1 and uses *Ortools* [7] as a solver to determine the scheduling routing path for each LC service request.

On the other hand, if $\sum_{s_i \in \mathcal{S}^k} t_i^k > 0$, it means that the total number of pending requests has exceeded the capacity of nodes to process them. In such a situation, *DSS-LC* uses the random sorting function $\rho(\cdot)$ to divide the requests into two groups: $\mathcal{R}_k$, which are expected to be processed immediately, and $\mathcal{R}_k'$, which must be queued. Note that the priority policy of $\rho(\cdot)$ can be changed as required (LC services are of the same priority as each other in our scenario). Consequently, the graphs $\mathcal{G}_k$ and $\hat{\mathcal{G}}_k'$ are constructed, respectively. The graph $\mathcal{G}_k$ is constructed based on $\mathcal{R}_k$, and the routing solutions are obtained using *Ortool*, as in the first case. However, for graph $\hat{\mathcal{G}}_k'$, *DSS-LC* takes into account the total resources of edge nodes to construct it while considering the heterogeneous nature of edges, as shown in Eq. 7:

$$t_{i,k}' = -\min(r_{i,\text{total}}^{c,\,k}/r_i^{c,k}, r_{i,\text{total}}^{m,\,k}/r_i^{m,k}) \cdot \lambda, \tag{7}$$

where λ is the augmentation factor which guarantees that the total number of requests in the graph $\hat{\mathcal{G}}_k'$ matches the number in the set $\mathcal{R}_k'$ as shown in Eq. 8:

$$\lambda = \frac{\sum_{s_i \in \mathcal{S}^k} t_i^k}{\sum_{s_i \in \mathcal{S}^k} \min(r_{i,\text{total}}^{c,k}/r_i^{c,k}, r_{i,\text{total}}^{m,k}/r_i^{m,k})} \tag{8}$$

After constructing $\mathcal{G}_k$ and $\hat{\mathcal{G}}_k'$, *DSS-LC* dispatches all requests by following the routing path generated by *Ortool*.

5.3 Request Scheduling of BE Services

We choose to use centralized scheduling for BE service requests because (1) these requests do not have strict QoS targets, allowing for an additional transfer to the central edge-cloud cluster; and (2) scheduling decisions based on the global state information of the edge-cloud system are more comprehensive than those for LC service requests, which are limited to geo-nearby clusters. This approach also helps to reduce decision conflicts.

Centralized scheduling of BE service requests, however, poses two challenges. First, the approach must handle the large-scale dynamic network topology and minimize its impact on system performance. Second, scheduling decisions must be optimized for long-term throughput rather than a single moment. To address these challenges, we propose a **Deep reinforcement learning Customized algorithm based on Graph neural network for centralized BE requests scheduling (*DCG-BE*)**. As illustrated in Figure 7 and Alg. 3, *DCG-BE* utilizes graph representation learning [16] with a Graph Neural Network (GNN) to encode the large-scale network topology for the first challenge, [6] and employs Deep Reinforcement Learning (DRL) to obtain the scheduling routing path for the second challenge.

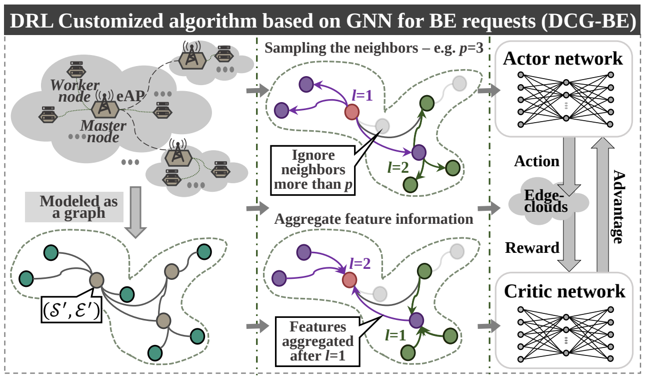

DRL Customized algorithm based on GNN for BE requests (DCG-BE)
Worker node
eAP
Master node
Modeled as a graph
(𝒮′, ℰ′)
Sampling the neighbors – e.g. p=3
l=1
l=2
Ignore neighbors more than p
Aggregate feature information
l=2
l=1
Features aggregated after l=1
Actor network
Action
Edge-clouds
Reward
Advantage
Critic network

Figure 7: Centralized BE service request scheduling.

[6] In our problem, GNNs outperform other graph representations based on sequences (e.g., recurrent neural networks) because GNNs remove the dependence on the order in which nodes are presented in the input [44].

5.3.1 *Markov Game Formulation.*

DCG-BE maintains a global graph $\mathcal{G}' = (\mathcal{S}', \zeta')$, where $\mathcal{S}'$ and ζ' represent the sets of nodes and edges, respectively. We formalize the scheduling problem of BE service requests as a Markov game $\mathcal{M} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, as shown below:

- **State $\mathcal{T}$.** The state of node s_i includes the following parameters: available CPU $r_i^{c,\text{ava}}$, available memory $r_i^{m,\text{ava}}$, total CPU $r_i^{c,\text{total}}$, total memory $r_i^{m,\text{total}}$, current slack score δ^k (mentioned in §4.3), CPU requirement of the BE request $r_i^{c,\text{be}}$, and memory requirement $r_i^{m,\text{be}}$. Similarly, the state of edge (s_i, s_j) includes the transmission latency $t_{i,j}^{\text{delay}}$ (between node s_i and s_j) and the requested transmission capacity $c_{i,j}$.
- **Action $\mathcal{A}$.** The action of *DCG-BE* at time t is defined as $a_t \in 1, 2, \ldots, N$, specifying the target node to which the request should be forwarded. And N denotes the number of nodes in the set $\mathcal{S}'$.
- **State transition probability $\mathcal{P}$.** *DCG-BE* employs the function $p\left(s_{t+1} \mid s_t, \boldsymbol{a_t}\right) : \mathcal{S}' \times \mathcal{A} \times \mathcal{S}' \rightarrow [0, 1]$ to represent the probability of executing the scheduling decision $\boldsymbol{a_t}$ and transitioning from state $\boldsymbol{s_t}$ to state $\boldsymbol{s_{t+1}}$.
- **Reward $\mathcal{R}$.** A comprehensive reward is designed to consider both *short-term and long-term availability of resources to improve long-term throughput and ensure load balancing* at the edge. The reward function is defined as $r_t = r_t^{\text{short}} + \eta \cdot r_t^{\text{long}}$, where η is a weight used to adjust the relative importance of the short-term and long-term rewards. Empirically, we set η to 1. The short-term reward r_t^{short} is defined as $r_t^{\text{short}} = e^{-\max(\sum_{\mathrm{q} \in Q_{\mathrm{t,i}}} \mathrm{r_q^c} / \mathrm{r_i^{c,node}}, \sum \mathrm{q} \in Q_{\mathrm{t,i}} \mathrm{r_q^m} / \mathrm{r_i^{m,node}})}$, where e is the Euler number, $Q_{t,i}$ is the set of BE service requests waiting to be processed in node s_i at time t, r_q^c and r_q^m denote the CPU and memory requirements of BE request q, and $r_i^{c,\text{node}}$ and $r_i^{m,\text{node}}$ denote the CPU and memory resources available on node s_i. The long-term reward r_t^{long} is defined as $r_t^{\text{long}} = 1 - e^{-\sum_{i=0}^{\hat{N}} \sum_{q' \in Q'_{t,i}} (r_{q'}^c / r_i^{c,\text{node}} + r_{q'}^m / r_i^{m,\text{node}})}$, where $\hat{N}$ is the number of actions between two training intervals and $Q'_{t,i}$ is the set of BE service requests completed in node s_i at time t.

5.3.2 *Algorithm Design of DCG-BE.*

GNN-Based network topology Encoding. We utilize a GraphSAGE network (Graph SAmple and aggreGatE) [18] to encode the large-scale network topology and create a graph embedding. GraphSAGE offers an inductive representation learning approach that is advantageous for large-scale graphs and is better suited to adjust to changes in topology [18]. The GraphSAGE network includes two primary steps: sampling and aggregation.

- **Sampling.** In the sampling step, a fixed number of samples p is set for each node $s_i \in \mathcal{S}'$ to improve computational efficiency. For each s_i, we perform sampling between neighboring nodes based on the indicator function $h(s_i, s_j)$, which is set to 1 if $(s_i, s_j) \in \zeta'$ (ζ' is the set of edges) and 0 otherwise. To select p neighbors for s_i, we check if $\sum_{s_j \in \mathcal{S}'} h(s_i, s_j) \leq p$; if not, we perform sampling without replacement after selecting p neighbor nodes. We define the set of neighboring nodes for node s_i as $\mathcal{N}(s_i)$.
- **Aggregation.** After selecting the neighboring nodes, the node aggregation is performed. Let $l \in \{0, 1, \ldots, L-1\}$ represent the index for the aggregation count, where we set the total aggregation count $L = 2$. We define the vector of node s_i at the l-th aggregation as v_i^l. The aggregation method can be expressed as follows:

$$v_i^{l+1} = \sigma(W \cdot \text{MEAN}(v_i^l \cup \{v_j^l, \forall s_j \in \mathcal{N}(s_i)\})), \tag{9}$$

where W is the training weight parameter and $\sigma(\cdot)$ is the activation function.

Algorithm 3: Intelligent BE service request scheduling (DCG-BE)

```
1  Initialize global parameters for central cluster;
2  if the BE scheduling queue is not empty then
3      for each BE service request do in parallel
4          Constructing the global graph $\mathcal{G}'$;
5          for the index of aggregation $l < L$ do
6              Sampling $p$ neighbors for each node;
7              Compute the vector $\mathrm{v}_i^{l+1}$ based on Eq. 9;
8          Compute $\hat{\boldsymbol{p}}(\hat{\boldsymbol{s}}_t)$ and obtain the action $a_t$;
9          Execute the action $a_t$ to get reward $r_t$;
10         if the required number of samples are collected then
11             Train and update parameters;
```

DRL-Based Centralized Scheduling. *DCG-BE* utilizes the Advantage Actor-Critic (A2C) algorithm [1] for its DRL part, consisting of an actor and a critic. The GNN resulting embedding vector, which captures the essential features of the topology, serves as the input for the actor. Based on this input, the actor generates a decision action a_t, while the critic evaluates the actor's decision. As detailed in §5.3.1, the action in *DCG-BE* is defined as the target node for which the BE service request should be scheduled. For requests that have reached the target node but cannot be processed in a timely manner, they will be returned to the scheduling queue for rescheduling.

To ensure the effectiveness of the action, we design a policy context filtering mechanism to filter out unavailable nodes. We define $\boldsymbol{c_t} \in \{0, 1\}^N$ to denote the validity of the scheduling decision. If the available resources of node s_i, i.e., $(r_i^{\text{c,ava}}, r_i^{m,\text{ava}})$ cannot satisfy request requirements, then $\boldsymbol{c_t} = 0$, otherwise $\boldsymbol{c_t} = 1$. Next, we define $\boldsymbol{p}(\boldsymbol{s_t})$ as the original output logits of the neural network based on the state $\boldsymbol{s_t}$. We then use $\boldsymbol{p}(\boldsymbol{s_t}) = \boldsymbol{p}(\boldsymbol{s_t}) * \boldsymbol{c_t}$ as the valid output logits, where * denotes element-wise multiplication. This ensures the validity of the output actions made by *DCG-BE*. In terms of implementation, both the actor and critic networks use a three-layer ReLU NN with 256, 128, and 32 hidden units per layer. The Adam optimizer with a fixed learning rate of 2×10^{-4} is used.

6 IMPLEMENTATION

We develop a *Tango* prototype, which comprises approximately 5000 lines of Python code. *Tango* is built on *Ubuntu* 20.04.3 LTS with *Linux kernel* v5.3.0-28, and it extends on *K8s* v1.21.0 by utilizing the *K8s-SDK* [4]. The *DCG-BE* algorithm is implemented using *PyTorch* 1.11.0. In order to emulate a realistic edge-cloud environment, we control the network bandwidth and the round-trip time (RTT) using the Linux Traffic Control tool, based on latency and geographic distribution information from the production dataset.

Note that *Tango* is designed with a **strong emphasis on asynchrony and parallelism**. We employ the *multiprocessing*, *ThreadPoolExecutor*, and *Lock* modules to enable the system to handle high

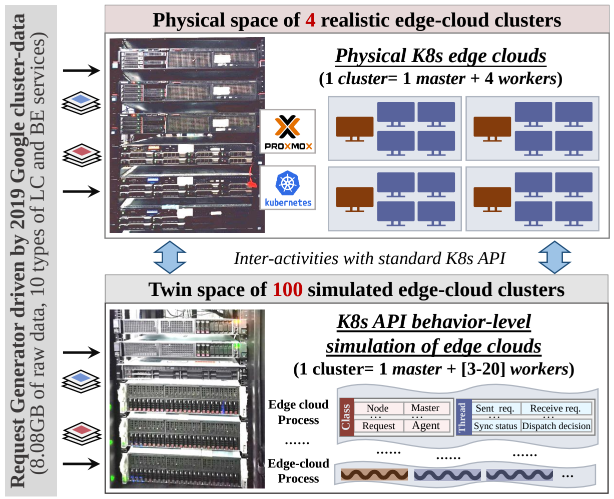

Request Generator driven by 2019 Google cluster-data
(8.08GB of raw data, 10 types of LC and BE services)
Physical space of 4 realistic edge-cloud clusters
Physical K8s edge clouds
(1 cluster= 1 master + 4 workers)
PROXMOX
kubernetes
Inter-activities with standard K8s API
Twin space of 100 simulated edge-cloud clusters
K8s API behavior-level simulation of edge clouds
(1 cluster= 1 master + [3-20] workers)
Edge cloud Process
Edge-cloud Process
Class
Node
Master
Request
Agent
Thread
Sent req.
Receive req.
Sync status
Dispatch decision

Figure 8: The dual-space edge-cloud experimental system with hundreds of mixed physical and virtual edge-clouds.

request-per-second (RPS) rates. Specifically, we have modularly subdivided *Tango*'s important functions, each of which is assigned to a corresponding process or thread pool, thereby avoiding the global interpreter lock (GIL) issue in Python.

6.1 Dual-Space Edge-Cloud System

A dual-space edge-cloud experimental system is illustrated in Figure 8. It comprises of four physical edge-cloud clusters, each composed of four worker nodes (with 4 CPUs and 8GB RAM) and one master node (with 8 CPUs and 16GB RAM). Additionally, we create a further 100 virtual edge-cloud clusters, supported by two servers, each with 64 vCPUs and 128GB RAM, in the simulation environment. To better reflect the heterogeneity of the edge, each virtual edge-cloud cluster consists of 3-20 *virtual worker nodes* represented as class objects, with a total of 1000 nodes.

For each virtual edge-cloud cluster, we create a system process to manage the lifecycle of requests. The virtual edge-clouds have the same logical operations as the physical edge-clouds but do not have physical container instances. To map the time taken to process requests in the simulation environment, we record the time taken for each type of service to complete under different loads and resources through pressure testing in the physical environment. We set an update thread for each virtual worker node, which wakes up periodically (every 100ms), to update and synchronize the current container resource status and the progress of request processing maintained in the form of a dictionary. Both physical and virtual edge-cloud clusters communicate through a TCP connection with a unified format for input and output.

6.2 Services and Workload Traces

The workload trace from the *2019 Google cluster-data* [2] is selected for our experiments. We extract records with the fields <EventType, SCHEDULE> and <CollectionType, JOB> from the dataset. The field "LatencySensitivity" is used to classify service types into 10 categories of LC and BE services. Each application is instantiated in a single container, and the expected resource allocation or QoS target (tail latency) is determined based on network transfer latency and service pressure measurement, leveraging methods mentioned in *PARTIES* [14]. Furthermore, we use one server as a *request generator* to send LC or BE service requests. In §7, we run each experiment five times, and *each period* in figures represents 800ms, which is the frequency at which we collect data.

7 EVALUATION

In the evaluation, we aim to answer the following questions:

- How does *HRM* perform when handling various loads (§7.1)?
- How do*Tango*'s two scheduling algorithms perform (§7.2)?
- Can *Tango* adapt to system scale expansion, and how does its performance compare to other state-of-the-art approaches under real workload traces (§7.3)?

7.1 *HRM* Effectiveness

First, we compare the performance of *K8s* with *Tango*'s *HRM* and *K8s-native*. We prepare three workloads with different characteristics (see Figure 9(a)): *(i)* pattern $\mathbb{P}_1$, which periodically sends LC service requests and randomly sends BE service requests; *(ii)* pattern $\mathbb{P}_2$, which periodically sends BE service requests and randomly sends LC service requests; *(iii)* pattern $\mathbb{P}_3$, in which both types of requests are sent randomly. We use the default policy of *K8s* as the scheduling algorithm for LC and BE service requests. We initialize the resource allocation limits of services for *K8s-native* according to the total resource usage ratio in the trace. Due to the flexibility of *D-VPA*, *Tango* does not need to be bound to a fixed allocation scheme. We conduct experiments on physical edge-cloud clusters.

As shown in Figure 9, *HRM* efficiently manages edge-cloud resources: when LC service workload is low, BE service fully utilizes the nodes' free resources (pattern $\mathbb{P}_1$); when LC service workload fluctuates significantly, *HRM* quickly and resiliently adjusts resource allocation (patterns $\mathbb{P}_2$ and $\mathbb{P}_3$). Overall, *HRM* effectively improves edge-cloud resource utilization (Figure 9(d)). In contrast, *K8s-native* cannot efficiently coordinate mixed services' resource utilization under different workloads due to fixed allocation and unordered competition (Figure 9(c)). We test *D-VPA* for average time taken to perform a single scaling operation, which is found to be 23ms. This represents a significant reduction in time compared

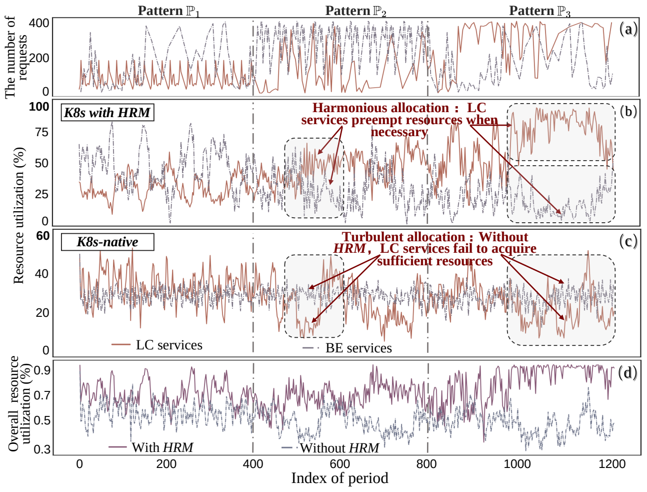


Figure 9: Three different requests workload patterns (a); the resource utilization of mixed services (b, c); comparison of the overall resource utilization (d).

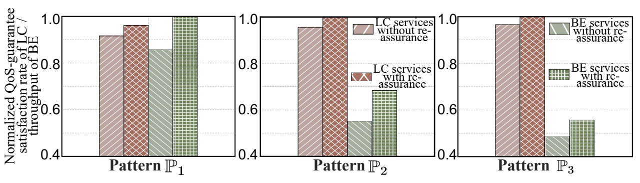


Figure 10: The performance of QoS-guarantee satisfaction rate and the throughput under three workload patterns.

to the delete-and-rebuild approach, by a factor of approximately 100 times. Note that this operation does not interrupt the running containers. We further verify the effectiveness of QoS re-assurance mechanism. Figure 10 shows that the QoS re-assurance mechanism effectively optimizes the system objective.

7.2 Scheduling Algorithm Performance

Based on the workload traces, we validate the scheduling algorithms' performance on physical edge-clouds.

Scheduling Algorithm for LC Service Requests. We compare *DSS-LC* against three other scheduling algorithms: *(i)* ***load-greedy***, which schedules requests to the node with the lowest load, *(ii)* ***K8s-native*** [9], the state-of-the-art default strategy used by *K8s*, and *(iii)* ***scoring*** [42], a weighted score algorithm that takes into account statuses such as resource usage and transmission latency. Additionally, we choose (ii) as the scheduling strategy for BE service requests. We consider three metrics for performance evaluation: QoS-guarantee satisfaction rate, tail latency (95^{th} percentile) of completed requests, and the number of abandoned requests.

Figure 11(a, b) illustrates that *DSS-LC* outperforms all other scheduling algorithms on all three metrics. The superiority of *DSS-LC* can be attributed to its construction of a graph structure data through classification, which integrates and optimizes the routing and forwarding of requests and queuing delays. Furthermore, *DSS-LC* exhibits the best stability in terms of QoS-guarantee satisfaction rate, indicating that the strategy effectively copes with fluctuating loads. *DSS-LC* is also ideal for **timely performance**, with a response time of 1.99ms for a node size of 500 and 3.98ms for a node size of 1000, which is less than 2% of the QoS target.

Scheduling Algorithm for BE Service Requests. To evaluate the performance of *DCG-BE*, we conduct a comparative analysis against several approaches, including *(i)* ***GNN-SAC***, an improved GNN-based learning algorithm that builds on the success of *SAC* [17]; *(ii)* ***load-greedy***; and *(iii)* ***K8s-native***, both of which are aforementioned. Similarly, We employ *K8s-native* as the scheduling strategy for LC service requests.

Figure 11(c) shows that all three inter-cluster scheduling algorithms outperform *K8s-native* by effectively utilizing system resources. Among them, *DCG-BE* and *GNN-SAC* optimize long-term throughput through global objective optimization. While *GNN-SAC* has strong exploration ability, it struggles to calculate strategy differences. In contrast, *DCG-BE* compares differences through online learning with the dominance function mechanism, leading to better performance. *DCG-BE* achieves 9.3% higher throughput than *GNN-SAC*. *Tango's DCG-BE* employs GraphSAGE to encode the large-scale network topology. We compare different GNN structures and find that GraphSAGE achieves the best performance by efficiently capturing essential topology features through inductive representation learning, as shown in Figure 11(d).

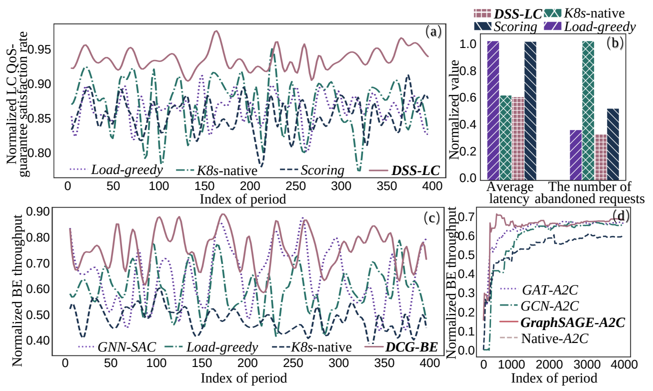


Figure 11: The comparison of *DSS-LC* with other algorithms in three metrics of LC services (a, b); the comparison of *DCG-BE* with others (c); the performance of *DCG-BE* to respond to different GNN structures (d).

7.3 Large-Scale Edge-Clouds Validation

Based on workload traces, we evaluate *Tango* in the large-scale hybrid (dual-space) edge-clouds which is described in §6.1.

Algorithm Pairing Analysis. We test different scheduling algorithm pairs under the same workload (Figure 12). When combined with any BE scheduling algorithm, *DSS-LC* outperforms others with approximately 8.2% higher rate of QoS-guarantee satisfaction. LC services are less affected by changes in BE scheduling policy due to *HRM*. However, the throughput of BE services significantly changes with LC scheduling policy. Among the LC scheduling algorithm combinations, *DCG-BE* achieves the best throughput performance. Moreover, when paired with *DSS-LC*, *DCG-BE* learns the most efficient scheduling strategy, surpassing the closest competitor (*DCG-BE* with *K8s*-native) by 5.9%. Thus, *DSS-LC* and *DCG-BE* form the optimal algorithm combination for *Tango*.

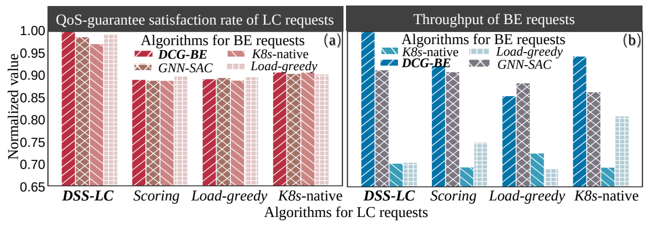


Figure 12: The QoS-guarantee satisfaction rate (a) and throughput (b) with combinations of scheduling algorithms.

State-of-the-Art Approach Comparison. We compare *Tango* to two state-of-the-art approaches: (i) *CERES* [40], a container-based elastic resource management system and (ii) *DSACO* [34], a distributed scheduling framework for edge computing based on SAC. *CERES* only provides a local resource management scheme, which cannot effectively utilize distributed and heterogeneous edge resources. Similarly, *DSACO* only provides an edge-oriented scheduling scheme, which cannot effectively manage resource allocation for mixed workloads. As shown in Figure 13, unlike *CERES* and *DSACO*, *Tango* can efficiently manage resources while scheduling mixed loads, leading to improved resource utilization. Specifically, *Tango* outperforms *CERES* by 36.9% in terms of resource utilization. Even though *DSACO* uses a intelligent distributed scheduling

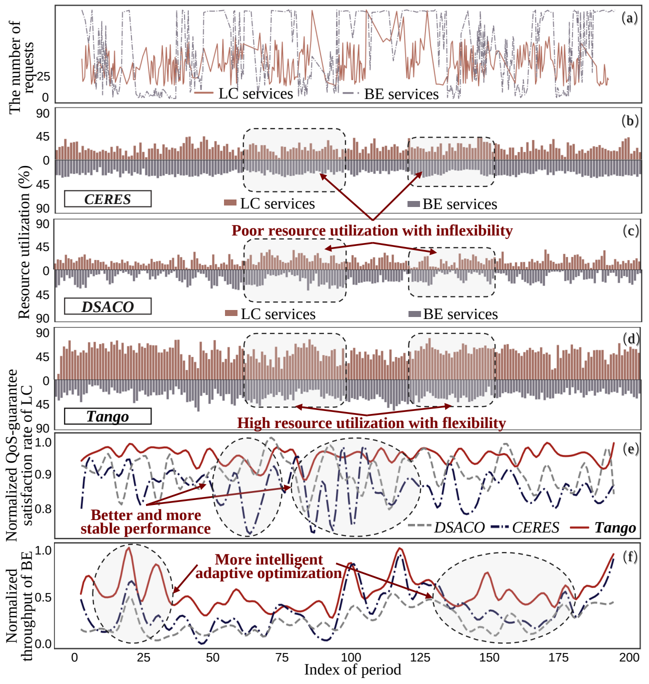


Figure 13: Comparing resource utilization (b, c, d), QoS-guarantee satisfaction rate (e), and long-term throughput (f) in large-scale hybrid clusters (a).

approach, *Tango's DSS-LC* with *HRM* support improves the QoS-guarantee satisfaction rate by 11.3% over *DSACO*. By incorporating *DCG-BE* with *HRM*, *Tango* achieves effective long-term throughput optimization over *CERES* by 47.6%.

8 CONCLUSION

In this paper, we have introduced *Tango*, a framework for managing and scheduling mixed services in *K8s*-based edge-cloud systems. *Tango* incorporates components, mechanisms, and two traffic scheduling algorithms to manage distributed resources. Our experiments have demonstrated that *Tango* dispatches BE requests while ensuring QoS for LC requests. Compared to state-of-the-art approaches, *Tango* improves resource utilization by 36.9%, QoS-guarantee satisfaction rate by 11.3%, and long-term throughput by 47.6%.

ACKNOWLEDGMENTS

This work was sponsored by the National Science Foundation of China under Grant No. 62072332, the China NSFC (Youth) through Grant No. 62002260, the Tianjin Xinchuang Haihe Lab under Grant No. 22HHXCJC00002, the Guangdong Pearl River Talent Recruitment Program under Grant No.2019ZT08X603, and the Guangdong Pearl River Talent Plan under Grant No. 2019JC01X235.

REFERENCES

[1] 2019. A2C. https://github.com/openai/baselines/tree/master/baselines/a2c
[2] 2021. Google data. https://github.com/google/cluster-data
[3] 2023. K8s-HorizontalPodAutoscaler. https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/horizontal-pod-autoscaler-v1/
[4] 2023. K8s-SDK. https://https://github.com/kubernetes-client/
[5] 2023. Koordinator. https://github.com/koordinator-sh/koordinator
[6] 2023. Kubernetes. https://github.com/kubernetes/kubernetes
[7] 2023. Ortools. https://developers.google.com/optimization/
[8] 2023. Prometheus. https://github.com/prometheus/prometheus
[9] 2023. scheduler. https://kubernetes.io/docs/concepts/services-networking/
[10] 2023. Swarm. https://docs.docker.com/engine/swarm
[11] 2023. VPA. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler
[12] Pradeep Ambati et al. 2019. Optimizing the cost of executing mixed interactive and batch workloads on transient vms. *POMACS* 3, 2 (2019), 1–24.
[13] Jun Lin Chen et al. 2022. Starlight: Fast Container Provisioning on the Edge and over the WAN. In *USENIX NSDI.*
[14] Shuang Chen et al. 2019. Parties: Qos-aware resource partitioning for multiple interactive services. In *ASPLOS.*
[15] Daniel E Eisenbud et al. 2016. Maglev: A fast and reliable software network load balancer. In *USENIX NSDI.* 523–535.
[16] Matthias Fey et al. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).
[17] Fujimoto et al. 2018. Addressing function approximation error in actor-critic methods. In *ICML.* 1587–1596.
[18] Will Hamilton et al. 2017. Inductive representation learning on large graphs. *Adv. Neural Inf. Process Syst.* 30 (2017).
[19] Rui Han et al. 2022. EdgeTuner: Fast Scheduling Algorithm Tuning for Dynamic Edge-Cloud Workloads and Resources. In *IEEE INFOCOM.*
[20] Yiwen Han et al. 2021. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. In *IEEE INFOCOM.*
[21] Lei Hindman et al. 2011. Mesos: A Platform for {Fine-Grained} Resource Sharing in the Data Center. In *USENIX NSDI.*
[22] Lei Huang et al. 2022. Towards Elasticity in Heterogeneous Edge-dense Environments. In *IEEE ICDCS.* 403–413.
[23] Yinzhi Lu et al. 2022. An Intelligent Deterministic Scheduling Method for Ultra-Low Latency Communication in Edge Enabled Industrial Internet of Things. *IEEE Trans. Industr. Inform.* (2022).
[24] Quyuan Luo et al. 2021. Resource scheduling in edge computing: A survey. *IEEE Commun. Surv. Tutor.* 23, 4 (2021), 2131–2165.
[25] Seyed Hossein Mortazavi et al. 2017. Cloudpath: A multi-tier cloud computing framework. In *ACM/IEEE SEC.* 1–13.
[26] Yoonsung Nam, Yongjun Choi, Byeonghun Yoo, Hyeonsang Eom, and Yongseok Son. 2020. EdgeIso: Effective Performance Isolation for Edge Devices. In *IEEE IPDPS.* 295–305.
[27] Rajiv Nishtala et al. 2020. Twig: Multi-agent task management for colocated latency-critical cloud services. In *IEEE HPCA.* 167–179.
[28] Parveen Patel et al. 2013. Ananta: Cloud scale load balancing. *ACM Comput. Commun. Rev.* (2013).
[29] Drew Penney et al. 2022. PROMPT: Learning Dynamic Resource Allocation Policies for Edge-Network Applications. *arXiv:2201.07916* (2022).
[30] Ju Ren et al. 2019. A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. *ACM CSUR* (2019).
[31] Shihao Shen et al. 2022. EdgeMatrix: A Resource-Redefined Scheduling Framework for SLA-Guaranteed Multi-Tier Edge-Cloud Computing Systems. *IEEE JSAC* (2022).
[32] Weisong Shi et al. 2016. Edge computing: Vision and challenges. (2016).
[33] Weisong Shi et al. 2016. Edge computing: Vision and challenges. *IEEE Internet Things J.* 3, 5 (2016), 637–646.
[34] Chuan Sun et al. 2021. Cooperative computation offloading for multi-access edge computing in 6G mobile networks via soft actor critic. *IEEE TNSE* (2021).
[35] Jianhang Tang et al. 2022. Latency-Aware Task Scheduling in Software-Defined Edge and Cloud Computing with Erasure-Coded Storage Systems. *IEEE Trans. on Cloud Comput. (Early Access)* (2022).
[36] Abhishek Verma et al. 2015. Large-scale cluster management at Google with Borg. In *ACM EuroSys.* 1–17.
[37] Jianyu Wang et al. 2019. Edge cloud offloading algorithms: Issues, methods, and perspectives. *ACM CSUR* 52, 1 (2019), 1–23.
[38] Xiaofei Wang et al. 2022. Integrating edge intelligence and blockchain: What, why, and how. *IEEE COMST* (2022).
[39] Mengwei Xu et al. 2021. From cloud to edge: a first look at public edge platforms. In *ACM IMC.* 37–53.
[40] Jinyu Yu et al. 2021. CERES: Container-Based Elastic Resource Management System for Mixed Workloads. In *ICPP.* 1–10.
[41] Ke Zhang et al. 2017. Mobile-edge computing for vehicular networks: A promising network paradigm with predictive off-loading. *IEEE Veh. Technol. Mag.* (2017).
[42] Yunqi Zhang et al. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *OSDI.*
[43] Zhiheng Zhong et al. 2020. A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources. *ACMTOIT* 20, 2 (2020), 1–24.
[44] Hang Zhu et al. 2021. Network planning with deep reinforcement learning. In *ACM SIGCOMM Conference.*