# Optimizing Resource Management for Shared Microservices: A Scalable System Design

SHUTIAN LUO*‡, University of Macau, China
CHENYU LIN‡, University of Macau, China
KEJIANG YE‡, Shenzhen Institute of Advanced Technology, CAS, China
GUOYAO XU, Alibaba Group, China
LIPING ZHANG, Alibaba Group, China
GUODONG YANG, Alibaba Group, China
HUANLE XU†‡, University of Macau, China
CHENGZHONG XU†‡, University of Macau, China

A common approach to improving resource utilization in data centers is to adaptively provision resources based on the actual workload. One fundamental challenge of doing this in microservice management frameworks, however, is that different components of a service can exhibit significant differences in their impact on end-to-end performance. To make resource management more challenging, a single microservice can be shared by multiple online services that have diverse workload patterns and SLA requirements.

We present an efficient resource management system, namely Erms, for guaranteeing SLAs with high probability in shared microservice environments. Erms profiles microservice latency as a piece-wise linear function of the workload, resource usage, and interference. Based on this profiling, Erms builds resource scaling models to optimally determine latency targets for microservices with complex dependencies. Erms also designs new scheduling policies at shared microservices to further enhance resource efficiency. Experiments across microservice benchmarks as well as trace-driven simulations demonstrate that Erms can reduce SLA violation probability by 5× and more importantly, lead to a reduction in resource usage by 1.6×, compared to state-of-the-art approaches.

CCS Concepts: • **Computer systems organization → Cloud computing**.

Additional Key Words and Phrases: Shared Microservices, Resource Management, SLA Guarantees

*S. Luo is affiliated with Yale University, but was at University of Macau during this work.
†Corresponding authors.
‡S. Luo, C. Lin, K. Ye, H. Xu and C. Xu are also with Guangdong-Hong Kong-Macao Joint Laboratory of Human-Machine Intelligence-Synergy Systems.

Authors' addresses: Shutian Luo, University of Macau, Macau SAR, China, stluo@um.edu.mo; Chenyu Lin, University of Macau, Macau SAR, China, MC14889@um.edu.mo; Kejiang Ye, Shenzhen Institute of Advanced Technology, CAS, Shenzhen, China, kj.ye@siat.ac.cn; Guoyao Xu, Alibaba Group, Hangzhou, China, yao.xgy@alibaba-inc.com; Liping Zhang, Alibaba Group, Hangzhou, China, liping.z@alibaba-inc.com; Guodong Yang, Alibaba Group, Hangzhou, China, luren.ygd@taobao.com; Huanle Xu, University of Macau, Macau SAR, China, huanlexu@um.edu.mo; Chengzhong Xu, University of Macau, Macau SAR, China, czxu@um.edu.mo.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0734-2071/2023/11-ART $15.00
https://doi.org/10.1145/3631607

Just Accepted

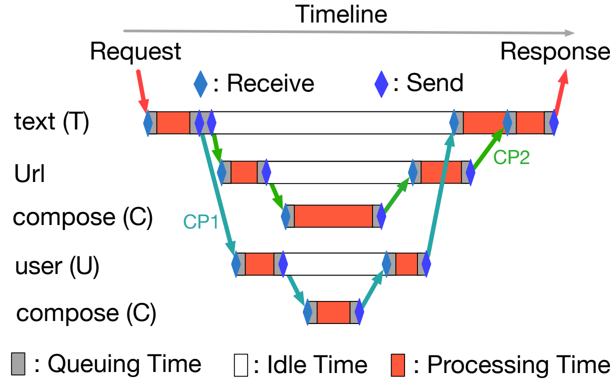



Fig. 1. A microservice *dependency graph* shows how microservice T interacts with other microservices. T calls *downstream* microservice Url and U in parallel and then calls Url and C one after another. The end-to-end latency measures the duration between T receiving the request and returning the result.

Just Accepted

## 1 INTRODUCTION

Recent years have witnessed a rapid emergence and wide adoption of microservice architecture in cloud data centers [4, 14, 15]. Compared to the conventional monolithic architecture that runs different components of a service within a single application, a microservice system decouples an application into multiple small pieces for ease of management, maintenance, and update [18, 25, 42, 42, 43, 48]. Due to this, microservices are light-weight and loosely-coupled. As a consequence, when microservice architectures are exposed to growing load, the system manager can locate individual microservices that may experience heavy load and scale them independently instead of scaling the whole application [13, 16].

Despite the flexibility, microservice architecture brings several new challenges in providing service-level agreement (SLA) guarantees for efficient resource management. First, a service request needs to be processed by hundreds of microservices [1, 28]. These microservices can form a complex dependency graph consisting of parallel, sequential and even alternative executions, as shown in Fig. 1. It becomes extremely difficult to manage resources at the granularity of microservices so as to maximize resource efficiency and in the meanwhile, ensure the end-to-end SLA. Second, microservice containers [10] are usually colocated with batching applications [26]. Resource interference can degrade differently the performance of microservices since some microservices are sensitive to resource interference. Third, resource interference can further cause performance imbalances between containers from the same microservice, especially when the workload is heavy, as a microservice usually contains hundred to thousands of containers.

Existing approaches provide SLA guarantees for microservice management via handcrafted heuristics, reinforcement learning approaches, or deep learning algorithms [7, 23, 36, 38, 41, 46, 47]. In particular, several heuristics adopt the average and covariance of microservice response time to determine the contribution that each microservice makes toward guaranteeing the end-to-end SLA requirement [23, 47]. One fundamental limitation of such solutions is that their derived contributions are fixed and do not change with the dynamic workload. The reinforcement learning approaches need substantial efforts for labeling critical microservices that have a great impact on SLA [38]. Moreover, when one service contains multiple critical microservices, independently keeping them tuning up can easily lead to sub-optimal results. Deep learning approaches need to evaluate a large number of potential resource configurations in order to find an efficient allocation without SLA violation [36, 46]. However, this is not scalable for complex services in production environments where a service can consist of 1000+ microservices with many tiers [28].

Furthermore, there is no study so far to investigate microservices sharing among different services with complex dependencies. However, shared microservices create a new opportunity to improve resource efficiency through global resource management among all services. To demonstrate this, we conduct a simple experiment to show that prioritizing services at a shared microservice can save more than 40% of resources (details are shown in § 2.3). As such, there is a crucial need for more efficient schemes that can globally manage SLAs for all services.

This paper addresses the aforementioned limitations by introducing Erms, a new system designed for efficient and scalable resource management in a shared microservice execution framework to provide SLA guarantees with high probability. Erms characterizes microservice tail latency in terms of a piecewise function of the workload, the number of deployed containers, and resource interference. With this characterization, Erms manages to dissect the detailed structure of microservice dependency graphs through explicit quantification and global optimization. This makes Erms fundamentally different from deep learning approaches [36, 46] and other heuristic solutions [23, 47].

Erms determines the latency target of each microservice so as to satisfy the end-to-end SLA requirement with minimum resource usage, based on the observed workload. At a shared microservice, Erms implements priority-based scheduling to orchestrate the execution of all requests from different online services. Under this scheduling, priority is given to services that include more latency-sensitive microservices, so as to significantly improve resource efficiency. Erms adopts a probability-based approach to implement priority scheduling, which can avoid potential starvation. Furthermore, Erms proposes a new interference-aware cluster-wide placement strategy aimed at balancing the latency across microservice containers and enhancing the overall performance of online services. Erms also incorporates careful designs to make the system scalable and applicable to production environments. The key techniques are in the application of convex optimization results and in the design of novel graph algorithms with low complexity.

We build a prototype of Erms on top of Kubernetes [24]. We evaluate Erms via real deployment on microservice benchmarks including DeathStarBench [18] and TrainTicket [49]. Additionally, we run large-scale simulations with real traces. Experimental results demonstrate Erms can reduce the number of deployed containers by up to 1.6× and reduce SLA violation probability by 5× compared to the state-of-the-art approaches. In summary, Erms has made the following contributions:

- **Optimal computation of microservice latency target**. To the best of our knowledge, Erms is the first system to systematically determine an optimal latency target for each microservice to meet SLA requirements. Erms is scalable to handle complex dependencies without any restrictions on graph topology.
- **New scheduling policy at shared microservices**. Another contribution of Erms is to design a new scheduling policy for shared microservices with theoretical performance guarantees. This policy assigns priority to requests from different services, and also globally coordinates resource scaling for all microservices. With this new policy, Erms can further reduce the number of used containers by up to 50%.
- **Implementation**. We provide a prototype implementation of Erms on top of Kubernetes [24], a widely adopted container orchestration system. We implement dynamic resource provisioning to place containers so as to control the overall resource interference.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Microservice Background

A production cluster often deploys various applications and each application contains multiple different online services to serve users' requests [29]. Usually, a service request is sent to an entering microservice, e.g., Nginx, which will then trigger a set of calls between multiple microservices. A microservice shall proceed to call its multiple *downstream* microservices either in a sequential manner or in parallel, when handling a call from its *upstream* microservice. Moreover, a microservice usually runs in multiple containers (with the same configuration)

Just Accepted

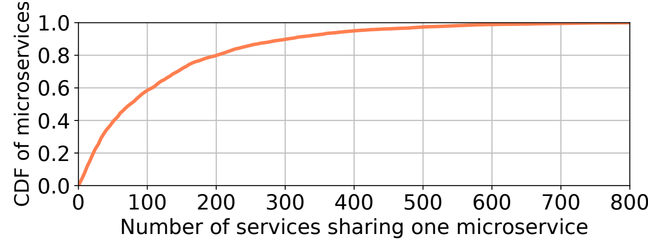



Fig. 2. The cumulative distribution of microservices shared by a different number of online services from Alibaba traces.

to serve all requests sent to it. In this paper, we adopt the number of containers as a metric for estimating resource usage, which is consistent with prior research [23, 38, 46, 47] and is widely recognized in the industry [28].

When handling a user request, the set of calls along with the associated microservices form a *dependency graph*. The performance of a user request, i.e., the end-to-end latency is determined by the longest execution time of all critical paths in the graph. Here, a critical path is a path that starts with a user request and ends with the service response to the corresponding request [38]. It is worth noting that a graph can contain multiple critical paths. For example, the *dependency graph* in Fig. 1 has two critical paths highlighted in blue and green colors respectively, $CP_1 = \{T, U, C\}$ and $CP_2 = \{T, Url, C\}$. In addition, the execution time on each critical path is the sum of all microservice latency along that path.

In addition to complex call dependency, microservices can also be multiplexed among multiple online services. We depict the degree of microservice sharing in Fig. 2 for traces collected from Alibaba clusters [1]. These traces include more than 20000 microservices and 1000 online services. Fig. 2 shows that 40% of microservices are shared by more than 100 online services. A shared microservice needs to process all requests from different services. When the workload of one online service (i.e., the request arrival rate) grows suddenly, the latency of requests from other services experienced at this microservice will increase significantly. Consequently, the end-to-end latency of one service can be greatly impacted by other services in a shared microservices execution framework.

## 2.2 Quantification of Microservice Latency

Compared to the end-to-end latency of online service, microservice latency is treated as a more fine-grained metric in terms of quantifying the resource pressure of deployed containers. Due to this, recent works begin to investigate how this performance metric can be affected by various factors such as the workload and resource interference on the physical host [7, 23, 47].

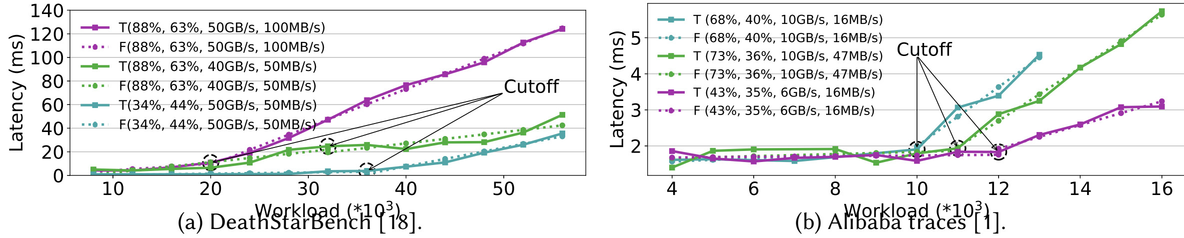
As shown in Fig. 1, the latency of a request at each microservice includes both the queuing time (in gray color) and processing time (in red color), which however, are difficult to obtain from a microservice tracing system since they require to probe the Linux kernel with high-overhead tools [19, 48]. By contrast, the timestamp of each *SEND* event and *RECEIVE* event of a request and a response in Fig. 1 is available from the tracing framework such as Jaeger [2]. Leveraging such information, we can derive the latency of a microservice by subtracting its *downstream* microservice response time from its own response time. More specifically, let $R_i^m$ and $S_i^m$ denote the timestamp that the $i$th request arrives at Microservice $m$ (aka *RECEIVE*) and the corresponding response leaves $m$ (aka *SEND*) respectively. When $d$ is the only *downstream* microservice of $m$, the latency of request $i$ at $m$ is:

$$L_i^m = (S_i^m - R_i^m) - (S_i^d - R_i^d). \tag{1}$$

If $m$ calls its multiple *downstream* microservices sequentially, each microservice's response time, i.e., $(S_i^d - R_i^d)$ should be subtracted from $(S_i^m - R_i^m)$ in Eq. (1). By contrast, if $m$ calls several *downstream* microservices in parallel, only the maximum response time of these microservice shall be subtracted from $(S_i^m - R_i^m)$. Note that, $L_i^m$ also includes the transmission latency, which can be obtained from the tracing system directly.



Just Accepted




(a) DeathStarBench [18]. (b) Alibaba traces [1].

Fig. 3. P95 Microservice latency from different traces (P99 behaves similarly). The four numbers in each bracket represent host resource usage, which include CPU utilization, memory capacity utilization, memory bandwidth usage, and network bandwidth usage. T is for ground truth and F is for fitting using a piece-wise linear function.

The study of microservice latency in existing works focuses on the first and second-order statistics across different workloads. In general, when the tail-latency of a microservice grows significantly in the workload, i.e., the call arrival rate, this microservice is considered to be critical for resource management. However, we observe from both existing microservice benchmarks [18] and Alibaba traces [1] that microservice always presents non-uniform delay performance when workload changes. As shown in Fig. 3, in each curve, each curve exhibits a distinct cut-off point (indicated by a black circle), which can be automatically determined as outlined in Section 5.2.1. Before reaching this cut-off point, the tail latency gradually and linearly increases with the workload. Conversely, once the workload surpasses this threshold, the latency of microservices experiences a considerably faster (almost linear) growth. The reason behind this is that each microservice container maintains a certain number of threads to process requests in parallel; therefore, when the workload is heavy and beyond a certain point, many requests need to be queued, resulting in a rapid increase in response time. As a result, current investigations on microservice latency are not meticulous and can easily lead to poor scaling decisions since they depend on a constant mean and variance [23, 47].

Another limitation is that, existing studies do not quantify the impact of resource interference on the slope of the latency curve [23, 46, 47]. Here, we measure resource interference in terms of resource usage of CPU, memory capacity, memory bandwidth and network bandwidth on physical hosts. Our quantification of microservice latency reveals that the slope changes when interference varies. As depicted in Fig. 3(a), when comparing a host with high resource usage (indicated by the purple line) to one with low resource usage (indicated by the blue line), the rate of increase in microservice latency after the cutoff is five times higher on the former. Additionally, resource interference causes the cut-off point to shift forward. In other words, as interference becomes more severe, microservice latency begins to increase rapidly at an earlier stage.

These observations motivate us to model microservice latency as a piecewise linear function of the workload. In addition, the slope of the linear curve highly depends on resource interference. With this function, we can quantify the performance of each microservice under different workloads and resource usages. It is possible to improve resource efficiency via globally optimizing resource configurations of all microservices based on the latency model and in the meanwhile, provide SLA guarantees with high probability.

To validate the above idea, we conduct a simple experiment for resource scaling in Fig. 4 where there is only one service consisting of two sequentially-executed microservices U and P from Social Network Application in DeathStarBench [18]. Based on the two profiled piece-wise linear functions and host utilization, we compute for both U and P a latency target, which specifies the maximum time each microservice can take to process a request to meet the end-to-end SLA. These two latency targets change with the service workload and their sum equals the end-to-end SLA. The details of the computation are described in § 4.2. U is given a higher latency target in contrast to P since its latency grows faster with the workload. The number of containers for U and P is then scaled

Just Accepted

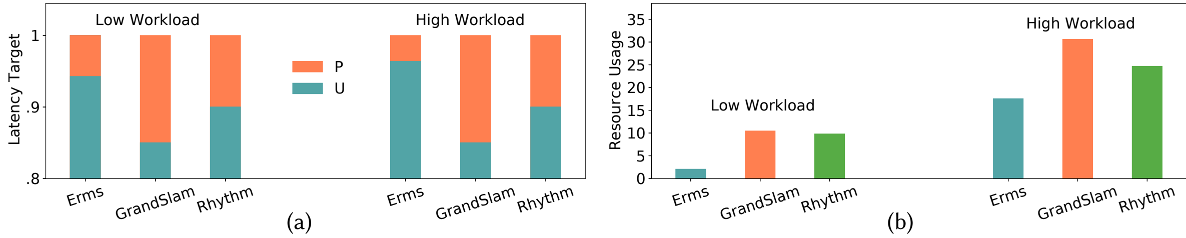



Fig. 4. An online service calls userTimeline (U) and postStorage (P) sequentially. The latency of Microservice U is more sensitive to workload changes than that of P. (a) Computed latency targets under different schemes in low-workload and high-workload settings. (b) The normalized total resource usage under different schemes.

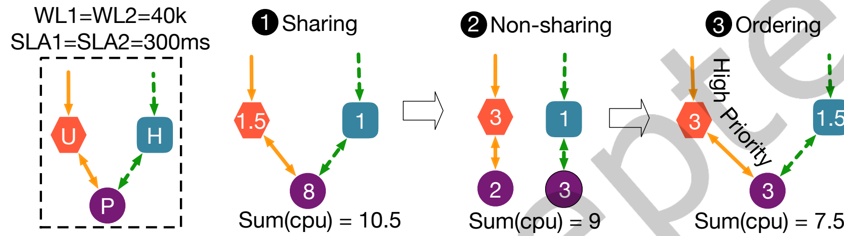



Fig. 5. Resource usage under microservice multiplexing for ensuring SLA requirement, the number in each circle represents the amount of CPU allocation to a microservice.

Just Accepted

such that the resulted microservice latency is below the corresponding target. Fig. 4(b) shows that this scaling can lead to a reduction of the number of deployed containers by up to 58% and 6× in heavy-load and light-load settings while keeping the same tail end-to-end latency, compared to heuristic approaches GrandSLAm [23] and Rhythm [47]. The reason behind is that baselines compute latency targets based on the mean of microservice latency, regardless of the workload and interference. Consequently, they tend to allocate a lower latency target to U in contrast to our result, thereby requiring much more containers to be deployed for U, as shown in Fig. 4(a).

## 2.3 Challenges and Opportunities from Microservice Multiplexing

As mentioned in § 2.1, an individual microservice can be multiplexed by hundreds of online services. However, services can form diverse *dependency graph*s and have different workload patterns. When these services perform scaling in a separate manner, their allocated latency targets at a shared microservice can vary a lot, simply taking the minimum latency target for scaling without differentiating services can lead to a waste of resources.

We construct a simple multiplexing scenario to demonstrate that efficient scheduling at a shared microservice is important. As shown in Fig. 5, this scenario consists of two online services that share a common microservice P (postStorage) from DeathStarBench. The first service calls U (userTimeline) and P sequentially while the second service calls H (homeTimeline) and P sequentially. Moreover, U is more sensitive to workload changes than H in terms of latency performance. To ensure a comprehensive and unbiased comparison of different resource allocation approaches, we explore a wide range of resource allocation configurations for microservices. We carefully select the configurations that minimize resource allocation while still satisfying the SLA requirements for each approach.

One straightforward solution is to process the concurrent requests that arrive at the shared microservice following the default policy FCFS (First-Come-First-Serve) and allocate a latency target in each service independently. Specifically, latency targets $T^U, T_1^P$ for U and P are allocated from the first service based on its SLA requirement $SLA_1$, and latency targets $T^H, T_2^P$ for H and P are computed based on the second SLA requirement $SLA_2$. To satisfy all SLA requirements, the final latency target for P is configured by taking the minimum between $T_1^P$ and $T_2^P$, i.e., $T^P = \min\{T_1^P, T_2^P\}$.

The second approach is to partition the deployed containers of P into two separate groups, one group serves the first service and the other group serves the second group. Under this non-sharing approach, the latency target is allocated in each group independently.

We run an experiment based on the constructed scenario to compare the resource usage under these two schemes above. In this experiment, we generate the same static workload (40k requests/minute) for two different services and set $SLA_1 = SLA_2 = 300ms$. Experimental results show the non-sharing scheme requires 9 cores (❷ in Fig. 5), whereas the sharing scheme (❶ in Fig. 5) requires 10.5 CPU cores to fulfill the SLA requirement. It seems that this result violates the rule that sharing should be more cost-effective than non-sharing since the former can fully utilize resources. We also build an M/M/1 queue to analyze the processing time at P under these two different schemes [20]. Indeed, the theoretical result validates sharing is better for the achieved mean processing time when fixing the resource usage. However, under resource scaling with SLA requirements, the bottleneck is the more-sensitive microservice, i.e., U in this scenario. Due to this, P is allocated a lower latency target in the first service. In the sharing setting, requests with a higher latency target (from the second service) can easily delay the processing of those with a lower latency target (from the first service). As a result, sharing leads to more resource usage under SLA-guaranteed scaling. This implies that the lack of global coordination in a shared microservice execution framework makes multiplexing inefficient, and it is better to process calls from different services separately. Nevertheless, this non-sharing scheme is inconsistent with the design principle of microservice architecture, i.e., microservice is designed to be loose-coupled and functionality-focused only.

To mitigate delay caused by less-sensitive microservices and improve resource efficiency, we design a priority-based scheduling policy under which requests from the first service are given higher priority at Microservice P (❸ in Fig. 5). Under this scheduling, latency targets need to be recomputed for microservices within the second service. The purpose of recomputation is to set a lower latency target for less-critical microservices, so as to relieve resource pressure on shared microservices. To examine this idea, we rerun the above experiment with the same workload and SLA settings. The result shows this policy only requires 7.5 CPU cores to satisfy SLA requirement, which is 20% (40%) less than that under the non-sharing scheme (FCFS policy). As such, multiplexing with efficient scheduling provides opportunities to greatly reduce the total resource usage, even in simple settings. However, globally coordinating all services is generally difficult when the number of shared microservices is large, which requires more careful designs.

Just Accepted

## 3 THE ERMS METHODOLOGY

In this section, we describe the overall architecture of Erms framework. Erms is a cluster-wide resource manager that periodically adjusts the number of containers deployed for each microservice, with the goal of meeting service SLAs while minimizing total resource usage.

Erms deploys a *Tracing Coordinator* (❶ in Fig. 6) on top of two tracing systems, Prometheus [3] and Jeager [2]. *Tracing Coordinator* generates microservice dependency graphs and extracts the individual microservice latency based on historic traces.

Erms includes an *Offline Profiling* module with two components, microservice Latency Profiling (❷ in Fig. 6) and Resource Usage Profiling (❸ in Fig. 6), which work in the background. This module fetches all microservice latency samples and resource usage samples under different workloads for all deployed containers for each

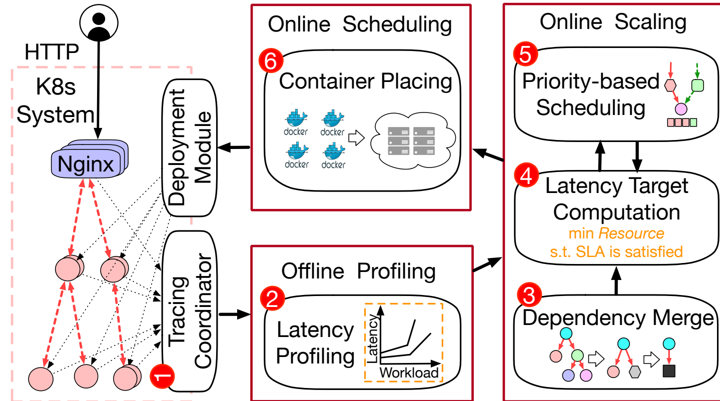



Fig. 6. The system architecture of Erms.

microservice from the *Tracing Coordinator*. With these data samples, microservice Latency Profiling builds a fitting model that profiles microservice tail latency as a piece-wise linear function of the workload. Additionally, using the collected samples, Resource Usage Profiling builds a linear model to estimate resource usage of microservice containers under different workloads.

The key module of Erms is *Online Scaling*, which makes scaling decisions according to workload changes. It consists of three components, i.e., Graph Merge (❹ in Fig. 6), Latency Target Computation (❺ in Fig. 6), and Priority Scheduling (❻ in Fig. 6). Graph Merge component applies graph algorithms to merge a general *dependency graph* with complex dependency into a simple structure with sequential dependency only, based on the observed workload. The purpose of this merge procedure is to simplify latency target computation. Latency Target Computation component allocates an initial latency target for all microservices within each *dependency graph* via solving a simple convex problem with low overhead. Priority Scheduling component assigns each service a different priority at a shared microservice based on this initial latency target. Requests from different services are processed according to this priority. Moreover, such priority also determines a new workload that a shared microservice needs to process under each service. Stem from this new workload, Latency Target Computation component recomputes latency targets for all microservices, and scales containers accordingly.

Erms also contains a *Container Placing* module (❼ in Fig. 6) to place all containers from different microservices across physical hosts in the cluster. This module places newly scheduled containers or release existing containers, which are determined by the *Online Scaling* module. The placement strategy aims to globally reduce the impact of resource interference on the end-to-end latency of online service. Specifically, the strategy takes into account the global resource interference within the physical hosts, which primarily arises from two sources: offline jobs and the microservices that are to be deployed on these hosts. Finally, actions are executed on the underlying Kubernetes cluster through the deployment module.

## 4 RESOURCE SCALING MODELS

In this section, we present the details of resource scaling models under Erms. First, we define the basic scaling model and our assumptions (§ 4.1). Next, we explain our developed solution approach and analyze why it works well (§ 4.2). The general principle behind this solution is to solve complex problems with near-optimality via using theoretically grounded yet practically viable solutions. Finally, we develop a multiplexing model to handle shared microservices (§ 4.3).

Just Accepted

## 4.1 Basic Model

Given a collection of service *dependency graph*s and all the microservices in each graph - together with quantified information about microservice latency and the workload relationship, and the container size of each microservice - we must deploy these services in the cluster such that their SLA requirements are satisfied, i.e., the tail end-to-end latency is smaller than a user-defined threshold while minimizing total resource usage. This yields the following optimization problem:

$$\min_{\overrightarrow{n}} \sum_{i=1}^{N} n_i \cdot R_i, \quad \text{subject to,} \;\; \text{latency}_k(\overrightarrow{n}) \leq \text{SLA}_k. \tag{2}$$

$\overrightarrow{n} = \langle n_1, n_2, \cdots, n_N \rangle$ is the decision vector where $n_i$ denotes the number of containers allocated to Microservice i. $N$ is the total number of unique microservices from all services. $R_i$ is the dominant resource demand of Microservice i, i.e.,

$$R_i = \max\left\{R_i^C/C \,,\; R_i^M/M\right\}, \tag{3}$$

where $R_i^C$ ($R_i^M$) is the size of CPU (Memory) configuration of containers from Microservice i, $C$ and $M$ are the overall CPU and Memory capacity in the cluster. $\text{latency}_k(\overrightarrow{n})$ and $\text{SLA}_k$ represent the tail end-to-end latency of requests from service $k$ under resource allocation $\overrightarrow{n}$ and the SLA requirement of service $k$, respectively.

As observed in § 2.2, microservice latency is a piece-wise linear function of the workload. For ease of modelling, we only consider a specific interval for each microservice in this section. In other words, the tail latency $L_i$ of Microservice $i$ is described as $L_i = a_i \frac{\gamma_i}{n_i} + b_i$. Here, $a_i$ and $b_i$ denote the slope and intercept, and $\gamma_i$ is the workload of Microservice i. The details of choosing intervals are presented in § 5.3.

## 4.2 Design of Optimal Scaling Method

In the setting where there is only one service consisting of sequential microservices, $\text{latency}_k(\overrightarrow{n})$ can be formulated as:

$$\text{latency}_k(\overrightarrow{n}) = \sum_{i=1}^{N} a_i \frac{\gamma_i}{n_i} + b_i. \tag{4}$$

In this setting, the optimal solution to Eq. (2) can be obtained via solving KKT equations corresponding to the convex optimization problem [6]. Consequently, the optimal latency target and the optimal number of containers $n_i^o$ can be expressed by a closed-form result:

$$a_i \frac{\gamma_i}{n_i^o} + b_i = \frac{\sqrt{a_i \gamma_i R_i}}{\sum_{i=1}^{N} \sqrt{a_i \gamma_i R_i}} \left(\text{SLA} - \sum_{i=1}^{N} b_i\right) + b_i. \tag{5}$$

Eq. (5) states that the optimal latency target of each microservice is in proportion to the square root of the product of $a_i$, workload $\gamma_i$, and resource demand $R_i$. This result implies that when the workload of a microservice increases, it needs to be allocated a higher latency target. Correspondingly, other microservices should be allocated lower latency targets and scheduled more containers.

A general *dependency graph* consists of multiple critical paths and one microservice can appear in different paths, complicating the optimal allocation of latency targets since it is difficult to give an exact expression of $\text{latency}_k(\overrightarrow{n})$. To address this problem, Erms simplifies the graph topology by removing parallel dependencies. We describe the procedure in Fig. 7, which shows how to merge parallel dependency within one *dependency graph* of workload $\gamma$. In Fig. 7, microservice T first calls microservice Url and U in parallel, and then calls microservice C after the response of Url and U.

**Extracting complete dependency graph**. In highly dynamic execution environments, dependency graphs within one service can vary significantly between each other. To address this issue, Erms compares the differences

Just Accepted

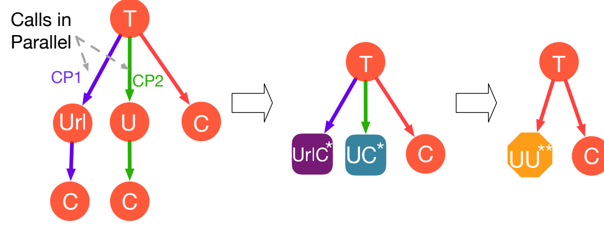



Fig. 7. Erms simplifies the structure of a general graph via gradually removing parallel dependency.

between dynamic graphs generated from the same online service and merges them into a complete dependency graph. Specifically, Erms first collects all microservices from historical traces and creates an empty zero matrix of size $\langle M, M \rangle$, where $M$ is the number of microservices, and each element represents an edge between two microservices. Erms then retrieves the dependency graph from the trace and updates the specific element from 0 to 1 if there is an edge between two microservices. This iteration is repeated until all graphs have been retrieved. Finally, the resulting adjacency matrix represents the complete dependency graph.

**Handling sequential dependency**. Erms removes dependency starting from the last layer, i.e., it first creates a virtual microservice UrlC* to merge Url and C, and creates another virtual microservice UC* to combine U and C. Let $\langle a_u, b_u \rangle$ and $\langle a_c, b_c \rangle$ be the parameters of the tail latency function associated with Url and C, and $\langle a_1^*, b_1^* \rangle$ and $\langle a_2^*, b_2^* \rangle$ be the parameters of UrlC* and UC*. The invention of a virtual microservice should yield the same latency and the same amount of resource usage as that of the original real microservices. Thus, the new parameters $\langle a_1^*, b_1^* \rangle$ can be characterized by:

$$a_1^* \frac{\gamma}{n_u + n_c} + b_1^* = a_u \frac{\gamma}{n_u} + b_u + a_c \frac{\gamma}{n_c} + b_c. \tag{6}$$

The solution to Eq. (6) is given by:

$$a_1^* = \left(\sqrt{a_u R_u} + \sqrt{a_c R_c}\right)\left(\sqrt{a_u / R_u} + \sqrt{a_c / R_c}\right), \tag{7}$$

$$b_1^* = b_u + b_c. \tag{8}$$

And the virtual resource demand of UrlC* is:

$$R_1^* = \left(\sqrt{a_u R_u} + \sqrt{a_c R_c}\right) / \left(\sqrt{a_u / R_u} + \sqrt{a_c / R_c}\right). \tag{9}$$

$\langle a_2^*, b_2^* \rangle$ can be obtained in the same way.

**Removing parallel dependency**. With the invention of UrlC* and UC* in Fig. 7, it remains to remove the parallel dependency between them. This can be achieved via inventing another virtual microservice UU**. Let $\langle a^{**}, b^{**} \rangle$ be the parameter of UU**. The optimal latency targets across parallel microservices must be the same, as otherwise, one can increase the lower one to reduce the overall resource usage. Thus, we have:

$$a_1^* \frac{\gamma}{n_1^*} + b_1^* = a_2^* \frac{\gamma}{n_2^*} + b_2^* \approx a^{**} \frac{\gamma}{n_1^* + n_2^*} + b^{**}. \tag{10}$$

The solution to Eq. (10) is as follows:

$$a^{**} = a_1^* + a_2^*, \; b^{**} = \max\left\{b_1^*, b_2^*\right\}, \tag{11}$$

and the virtual resource demand of UU** is given by:

$$R^{**} = (n_1^* R_1^* + n_2^* R_2^*) / (n_1^* + n_2^*). \tag{12}$$

After this merge process, the *dependency graph* only consists of three (virtual and real) microservices that execute sequentially. Erms computes latency targets and resource allocation for all these microservices based Eq. (5).



Just Accepted

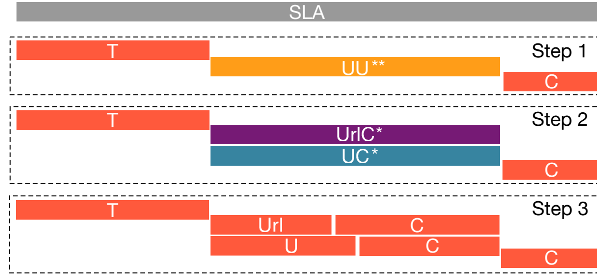



Fig. 8. An example of computing latency target for microservice graph in Fig. 7.

**Latency target computation**. Finally, Erms reverses the above graph merge procedure and computes a latency target for each microservice, as described in Fig. 8. First, Erms computes latency targets for microservices T, UU**, and C with sequential dependencies according to Eq. (5). Second, Erms assigns the same latency targets to microservices with parallel dependencies, that is, UrlC* and UC*'s latency targets are equal to UU**'s latency target. Last, Erms uses these results to compute latency targets for real microservices with sequential dependencies, i.e., $\{$Url,C$\}$ based on UrlC* and $\{$U,C$\}$ based on UC*.

Algorithm 1 describes the entire process of resource scaling with a general graph of known microservice characteristics and service workload. It adopts Depth-First Search (DFS) to find all two-tier invocations [28]. (Line 7 to Line 19). Each two-tier invocation consists of one microservice along with all its *downstream* microservices, e.g., $\{$T,Url,U,C$\}$ is a two-tier invocation formed by T, and $\{$Url,C$\}$ is another two-tier invocation formed by Url in Fig. 7. The merge function for inventing new virtual microservices (Line 24), starts from the last two-tier invocation and ends with the first one that is found by DFS. After this, the algorithm computes an optimal latency target for all virtual microservices (Line 20). The worst-case time-complexity of DFS algorithm is $O(|V| + |E|)$ for a graph with $|V|$ nodes and $|E|$ edges.

## 4.3 Microservice Multiplexing Model

Erms can extend the basic resource scaling framework to model multiplexing among different services.

Erms schedules high-priority services before those of low priority whenever there are multiple requests queued at a shared microservice. As such, response time of low-priority requests experienced at this shared microservice will be delayed by high-priority ones. To explicitly quantify such an effect, Erms formulates a new model to incorporate priorities assigned to different services. Consider two services illustrated in Fig. 5 with workload $\gamma_1, \gamma_2$ and SLA requirements $\text{SLA}_1$ and $\text{SLA}_2$ When requests from the first service are given higher priority at shared microservice P, and there is no other microservice shared among these two services, the new model is formulated as:

$$\sum_{i \in \Phi_1 \backslash \{p\}} a_i \frac{\gamma_1}{n_i} + b_i + a_p \frac{\gamma_1}{n_p} + b_p \leq \text{SLA}_1, \tag{13}$$

$$\sum_{i \in \Phi_2 \backslash \{p\}} a_i \frac{\gamma_1}{n_i} + b_i + a_p \frac{\gamma_1 + \gamma_2}{n_p} + b_p \leq \text{SLA}_2, \tag{14}$$

where $\Phi_1$ and $\Phi_2$ are the set of microservices included in the first and second services. In the first service, the end-to-end tail latency includes the time of processing $\gamma_1$ requests per unit of time at P. By contrast, for the shared microservice in the second service, its tail latency is the time to finish processing $(\gamma_1 + \gamma_2)$ requests. This model can be generalized to include more services multiplexing microservice P. It is worth noting that this problem is also convex with respect to the allocation vector $\overrightarrow{n}$.

Just Accepted

**Algorithm 1:** Resource Scaling Algorithm of Erms

```
1  unvisited: Stack for unvisited nodes;
2  twoTier: Stack for nodes in a two-tier invocation;
3  virtualNode: Stack for virtual microservices;
4  /* Depth-First Search                                        */
5  unvisited.push(EnterMicroservices);
6  parent = None;
7  while !unvisited.isEmpty() do
8      current = unvisited.pop();
9      if current.child is not Empty then
10         unvisited.push(current.child);
11         parent = current;
12     else
13         twoTier.push(current);
14         /* Two-tier invocations                               */
15         if parent == current then
16             v = Merge(twoTier);
17             Clear twoTier;
18             twoTier.push(v);
19             virtualNode.push(v);
20 while !virtualNode.isEmpty() do
21     M = virtualNode.pop();
22     /* Latency Target Computation                             */
23     Compute Latency Target for Microservice M with Eq. (5);
24 Function Merge(twoTier):
25     /* Merge Parallel Calls First                             */
26     if Calls in twoTier are parallel then
27         Create Virtual Microservice with Eq.(11);
28     /* Merge Sequential Calls Last                            */
29     Create Virtual Microservice with Eq. (7), (8);
30     return virtual Microservice;
31 End Function
```

Just Accepted

We also make use of convex analysis to quantify the total amount of resource usage under the multiplexing model. Theorem 1 demonstrates this new model results in less resource usage for satisfying SLAs, when compared to other scheduling policies.

Theorem 1. *The resource usage obtained by the optimization problem in Eq.* (13) *and Eq.* (14) *is smaller than that under the sharing scheme using FCFS scheduling and the non-sharing approach.*

In the following proof of Theorem 1, we empirically compare Erms' priority scheduling policy with other baselines, including sharing and non-sharing approaches, in terms of resource usage. The result demonstrates Erms' priority scheduling policy is more cost-effective than baseline schemes in ensuring SLA requirements.

Proof. When there is no prioritization with multiplexing, the service SLA requirements, $\mathrm{SLA}_1$ can be formulated as:

$$a_u \frac{\gamma_1}{n_u} + b_u + a_p \frac{\gamma_1 + \gamma_2}{n_p} + b_p \leq \mathrm{SLA}_1, \tag{15}$$

and $\mathrm{SLA}_2$ is the same as that in Eq. (14). We now consider a special setting where $\mathrm{SLA}_1 - b_u - b_p = \mathrm{SLA}_2 - b_h - b_p$. In this setting, the optimal resource allocation can be obtained by solving KKT equations that are similar to Eq. (4), resulting in a total amount of resource usage of:

$$RU^s = \frac{\left(\sqrt{a_u\gamma_1 R_u + a_h\gamma_2 R_h} + \sqrt{a_p(\gamma_1+\gamma_2)R_p}\right)^2}{\mathrm{SLA}_1 - b_u - b_p}. \tag{16}$$

When each service deploys microservice independently with no multiplexing, we can directly use the results in Eq. (5) to determine the optimal scaling for each microservice, which yields the following amount of resource usage:

$$RU^n = \frac{\gamma_1\left(\sqrt{a_u R_u} + \sqrt{a_p R_p}\right)^2 + \gamma_2\left(\sqrt{a_h R_h} + \sqrt{a_p R_p}\right)^2}{\mathrm{SLA}_1 - b_u - b_p}. \tag{17}$$

Applying Cauchy-Schwarz Inequality here, we have $RU^n \leq RU^s$ and the equality is attained if and only if $a_u R_u = a_h R_h$.

However, it is difficult to derive a closed-form solution to the problem formulated in Eq. (13) and Eq. (14). One approximation is to solve these two equations independently, which yields an upper bound for the total resource usage:

$$RU^o \leq \frac{\left(\sqrt{a_h\gamma_2 R_h} + \sqrt{a_p(\gamma_1+\gamma_2)R_p}\right)^2}{\mathrm{SLA}_1 - b_u - b_p} + a_u\gamma_1 R_u + \sqrt{a_u a_p R_u R_p}\gamma_1. \tag{18}$$

Moreover, it can be readily shown that the R.H.S. of Eq. (18) is less than $RU^n$. As such, we have $RU^o \leq RU^n \leq RU^s$. This completes the proof of Theorem 1. □

While this theorem can guarantee the optimality of Erms' scheduling policy, it does not quantify to what extent Erms can improve the baselines. The proof also implies that the actual improvement depends on the workload and the sensitivity of *upstream* microservice's response time to workload changes.

## 5 ERMS DEPLOYMENT

### 5.1 Tracing Coordinator

The tracing coordinator in Erms is developed based on two open-source tracing systems, Prometheus [3] and Jaeger [2]. Prometheus collects OS-level metrics including CPU and memory utilization for each microservice container as well physical hosts. Jaeger is a system to collect application-level metrics, including all calls send to each microservice and service response time. Jaeger adopts a sampling frequency of 10% to control the data collection overhead. It records two spans for each call between a pair of microservices; one starts with the client sending a request and ends with the client receiving the corresponding response, while the other starts with the server receiving the request and ends with it sending the response back to the client.

Tracing Coordinator extracts microservice dependency graphs based on historical traces from Jaeger. Specifically, it first treats the incoming microservice that receives user requests as the root node. If there is a call

Just Accepted

between two microservices, Tracing Coordinator adds an edge between them. In addition, if the client-side span of newly added calls overlaps the span of existing calls, those calls are marked as parallel calls, otherwise they are sequential calls. Tracing Coordinator repeats this process until it traverses all recorded calls. Based on the microservice dependency graph, Tracing Coordinator also extracts individual microservice latency.

## 5.2 Microservice Offline Profiling

In this subsection, we introduce Erms' offline profiling module in detail. Erms adopts a linear function model to profile microservice latency and container resource usage. And these profiling results are leveraged to facilitate efficient containers scaling (§ 4) and scheduling (§ 5.4).

*5.2.1 Latency offline profiling.* As explained in § 2.2, microservice latency can be described as a piece-wise linear function of workload. At the same time, resource interference can significantly impact the slope of the latency curve. Therefore, Erms primarily considers workload and resource interference when conducting the profiling of microservice latency [7, 27, 37, 38, 47].

In terms of interference, Erms mainly considers CPU utilization, memory capacity utilization, memory bandwidth utilization, and network bandwidth utilization of the physical host where the microservice container is located. As investigated in § 2.2, resource interference can have a significant impact on microservice latency [7, 27, 37]. Erms adopts machine learning methods to profile microservice latency in terms of workload and interference. Specifically, Erms collects the tail latency of all samples within the $j$th minute for each Microservice i from Tracing Coordinator, i.e., $L_i^j$. Erms also counts the total number of calls processed by each deployed container in the $j$th minute, i.e., $\gamma_i^j$. These two together with the average resource utilization are regarded as one data sample for Microservice i, i.e., $d_i^j = (L_i^j, \gamma_i^j, \mathrm{C}_i^j, \mathrm{MemC}_i^j, \mathrm{MemB}_i^j, \mathrm{N}_i^j)$ where the last four elements represent CPU, memory capacity utilization, memory bandwidth utilization, network bandwidth utilization respectively. Erms fits all these samples into a piece-wise model as shown below.

$$L_i^j = \begin{cases} (\alpha_i^1 \mathrm{C}_i^j + \beta_i^1 \mathrm{MemC}_i^j + \eta_i^1 \mathrm{MemB}_i^j + \delta_i^1 \mathrm{N}_i^j + c_i^1)\gamma_i^j + b_i^1, & \gamma_i^j \le \sigma_i, \\ (\alpha_i^2 \mathrm{C}_i^j + \beta_i^2 \mathrm{MemC}_i^j + \eta_i^2 \mathrm{MemB}_i^j + \delta_i^2 \mathrm{N}_i^j + c_i^2)\gamma_i^j + b_i^2, & \text{otherwise.} \end{cases} \quad (19)$$

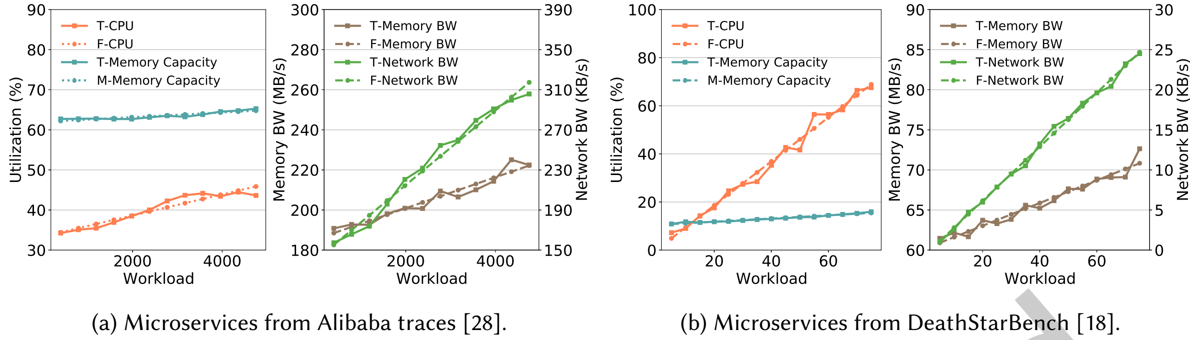
Provided there is resource interference, i.e., $\mathrm{C}_i^j$, $\mathrm{MemC}_i^j$, $\mathrm{MemB}_i^j$, and $\mathrm{N}_i^j$ remain fixed, $L_i^j$ can be portrayed as a piece-wise linear function of the workload $\gamma_i^j$. Consequently, Erms first iterates over all training samples with the same resource interference to identify the optimal one as the cut-off point $\sigma_i$ that minimizes the sum of squared residuals for the piece-wise linear function. Subsequently, using the least squares method, Erms fits the slopes $(a_i^l)_{l=1,2}$ and intercepts $(b_i^l)_{l=1,2}$ of the piece-wise linear function based on the optimal cut-off point $\sigma_i$. It is worth noting that $(b_i^l)_{l=1,2}$ are fixed values, unaffected by the resource interference.

Based on the fitted $a_i$ and $\sigma_i$, Erms proceeds to create a new training dataset for each microservice, capturing the impact of resource interference. In this training dataset, each element for Microservice $i$ consists of $\{\mathrm{C}_i, \mathrm{MemC}_i, \mathrm{MemB}_i, \mathrm{N}_i, \sigma_i, (a_i^l)_{l=1,2}\}$. To ensure efficient profiling, Erms employs simple yet effective models to quantify the relationship between $\sigma_i$, $(a^l)_{l=1,2}$ and the resource utilization $\{\mathrm{C}_i, \mathrm{MemC}_i, \mathrm{MemB}_i, \mathrm{N}_i\}$. Specifically, the slope $(a^l)_{l=1,2}$ is modeled as a linear function in relation to resource utilization. This means that the parameters $(\alpha_i^l, \beta_i^l, \eta_i^l, \delta_i^l, c_i^l)_{l=1,2}$ can be learned directly from the training dataset using the least-squares method. The cut-off point $\sigma_i$ is also a function of resource utilization, and Erms leverages a decision tree model [39] to learn this relationship.

*5.2.2 Resource usage profiling.* Microservices are mainly deployed to handle service requests, so the actual resource usage of microservice containers primarily depends on the service workload. As depicted in Fig. 9, both traces from Alibaba clusters and real benchmarks show that the average resource utilization of a running



Just Accepted




(a) Microservices from Alibaba traces [28].

(b) Microservices from DeathStarBench [18].

Fig. 9. Resource utilization of microservice containers grows linearly with microservice workloads.

container grows almost linearly with workload. Therefore, Erms adopts a linear regression model to profile container resource usage $r_i(\cdot)$:

$$r_i^l(x) = a_i^l \cdot w + b_i^l,\ l \in \{C, \mathrm{MemC}, \mathrm{MemC}, N\}, \tag{20}$$

where $w$ is the number of requests handled by per container within a minute for Microservice $i$, and $l$ presents four different hardware resources as described in Eq. (19).

## 5.3 Online Resource Scaling

In this section, we present the design details of *Online Scaling* module. The key of this module is to carefully apply resource scaling models developed in § 4 such that the scaling overhead is well controlled.

*5.3.1 Dependency merge and latency target computation.* Erms averages the current resource utilization across all physical hosts and feeds this utilization into the microservice profiling model to obtain parameters that describe the piece-wise linear function. These parameters quantify the sensitivity of microservice latency with respect to the workload of each container. Erms relies on them to allocate latency targets for microservices following Algorithm 1.

One critical challenge herein, however, is that there exist two different sets of parameters associated with two intervals for one microservice described by the profiling model. It is difficult to optimally choose which set should be used for Latency Target Computation. Exhaustively trying all possible choices is not scalable since the number of candidates is $2^m$ where $m$ is the number of microservice in a graph. To address this challenge, Erms first performs dependency merge and allocates latency targets based on these parameters learned from the second interval, as this interval corresponds to a high workload and means less resource consumption. After allocating a latency target for each Microservice i, Erms then checks whether the allocated latency target is less than the latency corresponding to the cut-off point $\sigma_i$ or not. A positive result means Microservice i requires extra resources and should be allocated a lower latency target. For these microservices, Erms adopts the other set of parameters in the first interval to recompute all latency targets. In this way, the *dependency graph* of each service needs to be processed at most twice for Latency Target Computation.

*5.3.2 Priority scheduling.* At a shared microservice, Erms needs to configure the scheduling priority of requests from different online services. To find the schedule that yields the fewest resource usage, it is required to solve the multiplexing model in § 4.3 under all possible configurations. However, this is not tractable in practical systems since there are $n!$ orderings if $n$ services share a microservice. When considering the situation that many microservices can be multiplexed among different services, the computational overhead can be extremely high, without mentioning the complexity of the multiplexing model. To be more scalable, Erms first calls the

Just Accepted

Table 1. Notations for placement optimization under Erms

| | |
|---|---|
| $c_i^l$ | The interference coefficient of resource $l$ for microservice $i$ (Eq. (19)) |
| $r_i^l$ | The usage of resource $l$ for Microservice $i$'s containers |
| $b_h^l$ | The usage of resource $l$ of existing jobs in Host $h$ |
| $R_l^c$ | The capacity of resource $l$ for Host $h$ |
| $p_{i,k}^h$ | Whether the $k$th container of $i$ is placed on $h$ |
| $n_i$ | Number of containers scheduled for Microservice $i$ |
| $\Psi$ | The set of four kinds of resource, including CPU, memory capacity, memory bandwidth and network bandwidth, respectively |
| $\Omega$ | The set of all Microservice |
| $\Phi$ | The set of all physical hosts |

Just Accepted

Latency Target Computation component for each service to allocate an initial latency target to all microservices. Priority is configured based on this target. In particular, the service that yields the lower latency target at a shared microservice is given higher priority. The intuition behind this is that the lower latency target implies the corresponding service consists of many latency-sensitive microservices and their requests should be handled first.

Based on the configured priority, Erms recomputes microservice latency target via solving the multiplexing model. However, this model couples all services together and is computationally expensive to deal with. For reducing scaling overhead, Erms chooses to call the Latency Target Computation component for each service independently. This call returns the final latency targets of all microservices and the number of containers to be scaled. In this call, Erms adopts a modified workload for a shared microservice to take into account priority scheduling. More specifically, let $\gamma_{k,i}$ denote the original workload at shared microservice i that is from service $k$, the modified workload is $\sum_{l=1}^{k} \gamma_{l,i}$, assuming services are ordered following their index. The result from Latency Target Computation implies when the workload of a microservice increases, other microservices within the same *dependency graph* should be set lower latency targets for resource efficiency. Based on this, Priority scheduling allocates more resources to non-shared microservices in order to relieve resource pressure on shared microservices, compared to FIFO scheduling.

Whenever a thread is available in a deployed container and there are requests waiting to be processed, a request from the service with higher priority will be assigned to this thread with higher probability. In particular, requests from the service with the highest priority are scheduled with probability $(1-\delta)$, and requests from the service with the $l$th highest priority are scheduled with probability $\delta^{l-1}(1-\delta)$, and the service with the lowest priority is scheduled with probability $\delta^{n-1}$ where $n$ is the number of services. Here, a small $\delta$ is beneficial to the response of high-priority services at the cost of starving the processing of low-priority requests when the workload is heavy. We shall evaluate the impact of $\delta$ on shared microservices in § 6.4.2.

*5.3.3 Overhead of resource scaling.* By careful design, Erms only needs to call Latency Target Computation twice for each *dependency graph*. In addition, Latency Target Computation component also applies graph traversal algorithm twice to compute latency targets, yielding a complexity of $O(|V|+|E|)$ for a graph with $|V|$ nodes and $|E|$ edges. In production clusters, *dependency graph*s behave like a tree [28], and the number of edges is usually several times the number of nodes. As such, the computational overhead of resource scaling scales linearly with the total number of microservices included in all services.

### 5.4 Interference-aware Containers Scheduling

To improve scalability, the Online Scaling module takes into account only the average resource interference across multiple hosts when performing resource scaling. However, it is important to note that scheduled containers belonging to a single microservice may be deployed across different hosts, resulting in varying degrees of resource interference. This variation in interference can subsequently lead to significant performance imbalances among containers within the same microservice.

A simple method to tackle performance imbalance involves bridging the disparity in host resource utilization[30]. However, this approach neglects the potential impact of resource interference on the performance degradation of different microservices to varying extents. In contrast, Erms strategically places containers in response to performance degradation in order to minimize end-to-end latency. To attain the optimal container placement, we develop an optimization problem with the objective function of minimizing the aggregate latency of all microservices (per eq. (19)). It is worth noting that this objective function accounts for resource interference originating not only from offline jobs but also from the microservices that will be placed on the hosts. The formulation of this optimization is as follows:

$$\min_{\boldsymbol{p}} \sum_{h\in\Phi}\sum_{i\in\Omega}\sum_{k=1}^{n_i} p_{i,k}^h \left\{ \Big( \sum_{l\in\Psi} \big(c_i^l \cdot (\sum_{i\in\Omega}\sum_{k=1}^{n_i} p_{i,k}^h \cdot r_i^l + b_h^l)/R_h^l\big) \Big) \gamma_i^j + b_i \right\} \tag{21}$$

$$\text{s.t.} \sum_{h\in\Phi} p_{i,k}^h = 1, \ \ \forall i,k \quad \text{and} \ \ p_{i,k}^h \in \{0,1\}, \ \ \forall i,k,h, \tag{22}$$

$$\sum_{i\in\Omega}\sum_{k=1}^{n_i} n_i r_i^l \le R_h^l, \ \ \forall h,l. \tag{23}$$

The explanation for each parameter can be found in Table 1, and the last constraint arises from the fact that the combined resource consumption of all containers on each host must not exceed the host's capacity. The resource usage of host $h$, as quantified in the objective function in Eq. 21, comprises two parts: usage from microservice containers to be deployed, $\sum_{i\in\Omega}\sum_{k=1}^{n_i} p_{i,k}^h \cdot r_i^l$ and usage from existing jobs, $b_h^l$. Given the workload $\gamma_i^j$, resource usage of microservice containers can be estimated based on eq. (20), while the resource usage of existing jobs can be retrieved through Erms's *Tracing Coordinator*.

It is worth noting that in this problem, $\boldsymbol{p} = \{p_{i,k}^h\}_{i,k,h}$ serves as the sole optimization variable. In the meanwhile, this problem is a non-linear integer programming problem, which is NP-hard and challenging to solve. To address this, we relax the integer constraint $p_{i,k}^h \in \{0,1\}$, allowing $p_{i,k}^h$ to assume a fractional number, i.e., $\widehat{p}_{i,k}^h \in [0,1]$. As a result, the problem transforms into a convex program, which can be efficiently solved using the ADMM approach [22]. Following this, the generated fractional solutions are rounded back to binary values through uniform random sampling, i.e., $p_{i,k}^h$ equals one with a probability of $\widehat{p}_{i,k}^h$. A significant limitation of this method is its high complexity, particularly when a production cluster contains a vast number of hosts and microservices. This complexity may result in substantial scheduling overhead, thereby restricting the approach's applicability. To alleviate this overhead, Erms statically divides a cluster's hosts into multiple equal-sized groups and solves a considerably smaller-scale optimization problem using the computational resources within each group.

Globally optimizing the placement of all containers may lead to migrations of containers across hosts. To mitigate the migration overhead, Erms solves the optimization problem based on the current deployment of containers in the cluster. If Erms determines to scale out the number of container for microservice $i$ from $n_i$ to $n_i^*$, then it only needs to figure out the placement for these $(n_i^* - n_i)$ containers.

Just Accepted

### 5.5 Erms Implementation

We implement a prototype of Erms on top of Kubernetes [24], a widely-adopted container orchestration framework. At runtime, Erms queries Prometheus to obtain real-time data for scheduling resources. *Online Scaling* module and *Resource Provisioning* module are written via Kubernetes Python client library, implemented in approximately 3KLOC of Python.

Erms implements the priority-based scheduling in the network layer of each container. More specifically, it relies on a Linux traffic control interface `tc` to manage different incoming network flows of a container. This interface can provide prioritization through a queuing discipline, i.e., `pfifo_fast`. As such, Erms only needs to specify the priority of each flow. Originally, `tc` is designed for controlling outcoming traffic rather than incoming traffic. Erms activates a virtual network interface in a physical host and then binds this interface to the desired container.

## 6 EVALUATION OF ERMS

### 6.1 Experiment Setup

**Benchmarks:** We evaluate Erms using an open-sourced microservice benchmark, DeathStarBench [18] and TrainTicket [49]. DeathStarBench consists of Social Network, Media Service, and Hotel Reservation applications. These applications contain 36, 38, and 15 unique microservices respectively, and include 3, 1, and 4 different services. Moreover, both Social Network application and Hotel Reservation application have 3 shared microservices. TrainTicket application contains about ten services, such as ticket booking, ticket querying and so on, and these services form dynamic dependency graph in runtime. Moreover, there are 23 shared microservices in these services.

**Cluster Setup:** We deploy Erms in a local private cluster of 20 two-socket physical hosts. Each host is configured with 32 CPU cores and 64 GB RAM. Each microservice container is configured with 0.1 core and 200MB memory.

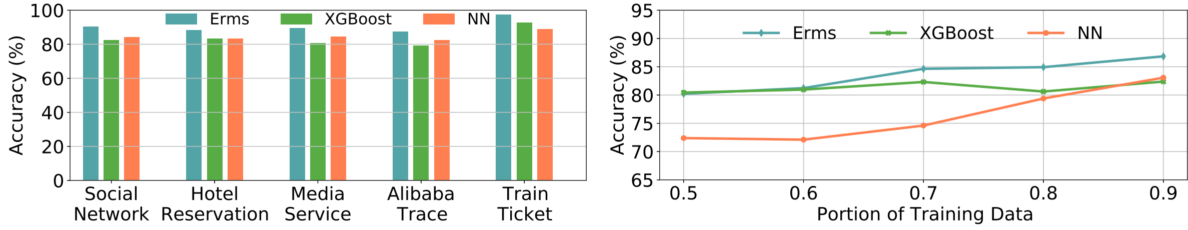
**Workload Generation:** We find that 100,000 requests reach the maximum throughput that our cluster can support in one minute for the benchmark [18]. As such, we generate multiple static workloads ranging from 600 (low) to 100,000 (high) requests per minute for each service. In addition, we also adopt dynamic workloads from Alibaba clusters [28]. SLA targets are set with respect to 95th percentile end-to-end latency, ranging from 50 ms (low) to 200 ms (high) for all applications.

**Dependency Graph:** In DeathstarBench, online services generally exhibit static dependency graphs while processing various requests. However, TrainTicket presents dynamic dependency graphs at runtime, influenced by distinct request arguments, such as the number of stations involved. We explore two representative TrainTicket services: ticket booking and ticket querying. Under the arguments for 1 and 10 stations, we generate simple and complex graphs, respectively. The combination of simple and complex graphs serves to highlight the dynamic nature of these dependency graphs.

**Baseline Schemes:** We compare Erms against GrandSLAm [23], Rhythm [47], and Firm [38]. Moreover, we include the original Erms' implementation (Erms-IPM) [30] as an additional baseline scheme. Without special mention, we set $\delta$ to 0.05.

- **GrandSLAm:** It computes latency target for each service such that it is proportional to its average latency under different workloads.
- **Rhythm:** It evaluates the contribution of each microservice as the normalized product of mean latency, and variance of latency across different workloads, as well as the correlation coefficient between microservice latency and the end-to-end service latency.
- **Firm:** It first identifies a critical microservice on each critical path that has a heavy impact on the end-to-end latency, and then applies reinforcement learning to tune resource allocation for this microservice.

Just Accepted




(a) Profiling accuracy under different APP and Alibaba traces. (b) Profiling accuracy using different portions of training data.

Fig. 10. Profiling accuracy using different algorithms on DeathStarBench and Alibaba traces.

Just Accepted

- **Erms-IPM:** To mitigate performance imbalances between containers of microservices, it minimizes the gap in resource utilization across hosts through container placement.

## 6.2 Microservice Profiling Accuracy

To validate the accuracy of Erms' microservice profiling module, we run DeathStarBench and TrainTicket in our local cluster and collect one-day running samples for each microservice. We fix the interference level on each host via injecting iBench workloads [11] during each hour, and collect one sample per minute for a microservice. In addition, we collect one-day samples for all microservices from Taobao Application in Alibaba traces [1]. Taobao is mainly for online shopping and it consists of 2000+ microservices. It is worth noting that microservices are usually co-located with batch jobs on the same host to increase resource utilization in Alibaba clusters [28]. Therefore, Alibaba microservices tend to experience more different types of resource interference than microservices in a dedicated cluster.

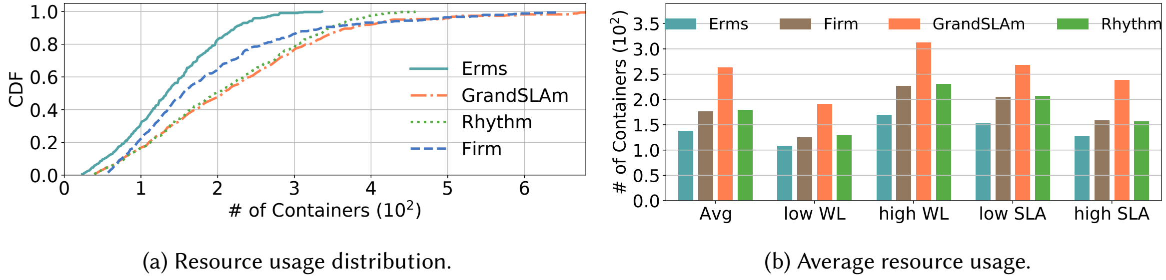
We train Erms' profiling model for each microservice using the first 22-hour samples and perform testing on the remaining samples. We also implement XGBoost [8] and a three-layer Neural Network (NN) with 64 neurons as baseline schemes. As shown in Fig. 10(a), the testing accuracy under Erms ranges from 83% to 97% for microservices from both DeathStarBench [18] and Alibaba traces. In this case, the testing accuracy is similar across all schemes. To investigate the generalization ability of Erms, we also evaluate the testing accuracy under different sizes of training data set collected from Taobao. As shown in Fig. 10(b), Erms achieves a testing accuracy of 85% using 70% of the training samples. In contrast, the testing accuracy under NN drops dramatically when the number of training samples reduces. Considering that Erms only needs the slope and intercept of a piecewise linear function for resource scaling, this testing accuracy is sufficient for resource management, even in production environments.

Moreover, we also evaluate the profiling results of resource usage using traces generated from DeathStarBench and TrainTicket, which collectively comprise nearly 120 microservices. Furthermore, we validate the efficiency of the linear regression model on more than 1000 microservices from Alibaba clusters. The results highlight that the prediction accuracy under these benchmarks and Alibaba traces can be as high as 92.2% and 91.2%, respectively.

## 6.3 Resource Efficiency and Performance

*6.3.1 Static workload.* In this part, we evaluate the resource usage and end-to-end latency of services under different static workloads and SLA settings. In each setting, we run all services for 30 minutes.

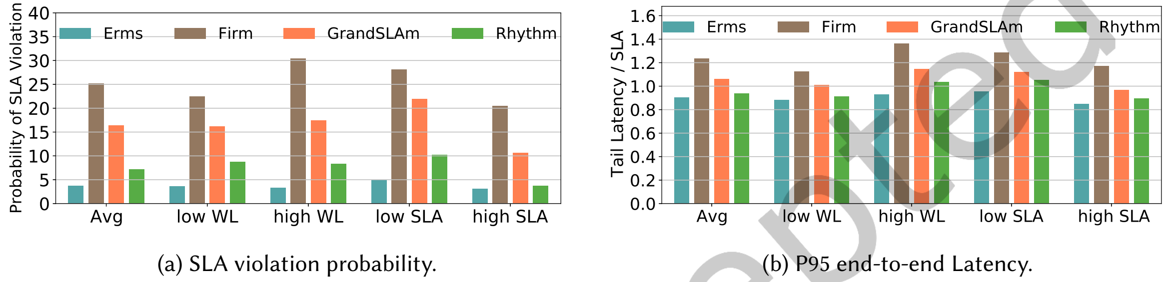
We quantify resource usage in terms of the number of containers allocated to all services. Fig. 11(a) shows the distribution of the resource usage under different static workloads. The result reveals that more than 83% of workloads require less than 200 containers under Erms, while these workloads need about 310 containers under both GrandSLAm, and Rhythm. GrandSLAm and Rhythm have similar distributions of resource usage as




(a) Resource usage distribution.

(b) Average resource usage.

Fig. 11. Containers allocated with static workloads.




(a) SLA violation probability.

(b) P95 end-to-end Latency.

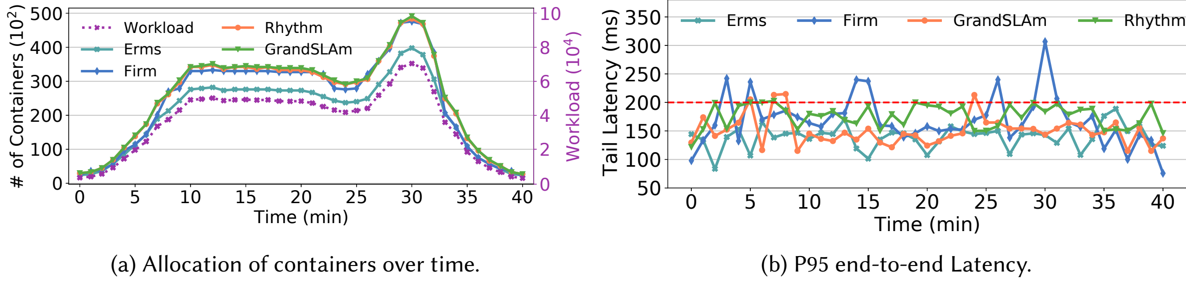
Fig. 12. Tail latency under different schemes.

Just Accepted

they allocate resources based on statistics of microservice latency. Firm tends to tune resource configuration for critical microservices only, and it needs to allocate more resources under the high workload to ensure SLA. As a result, Firm leads to the longest tail in term of the CDF distribution of resource allocation, as shown in Fig. 11(a). In an extreme case, Firm needs more than 3× resources compared to Erms. To be more comprehensive, we also compare these schemes in each specific setting, as shown in Fig. 11(b). On average, Erms saves about 27.8%, 91.1% and 30.1% of containers in contrast to Firm, GrandSLAm, and Rhythm, respectively. As workload goes up, the improvement of Erms also grows. One key reason behind this is that shared microservices need to deploy more containers so as to handle requests from different services, especially when the workload is high. This gives more opportunities for Erms to optimize resource allocation. Similar behavior can be observed when we vary SLA requirements. In the low-SLA scenario, the reduction of resource usage under Erms is more significant than that under the high-SLA setting. Low SLA means a low latency target allocated to each microservice and therefore, there is a large room to optimize resource usage.

In the meanwhile, we also characterize the end-to-end performance of service requests under different scenarios. As shown in Fig. 12(a), on average, the SLA violation probability under Erms is less than 4%, whereas it is as high as 25.2%, 16.4%, and 7.2% under Firm, GrandSLAm and Rhythm respectively. Moreover, both higher workloads and lower SLAs lead to higher SLA violation probability under all schemes. When referring to the actual end-to-end delay, Erms can reduce this metric by 18% compared to other schemes, as depicted in Fig. 12(b). Moreover, in the high workload and low SLA scenarios, the gap between end-to-end latency and SLA will be larger than that in the low workload and high SLA settings.

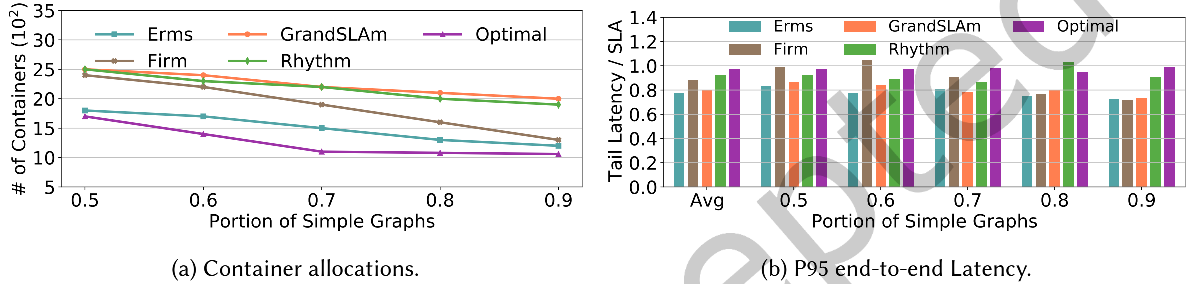
*6.3.2 Dynamic workload.* In this part, we generate dynamic workload based on Alibaba traces and set the SLA target to 200ms. In this experiment, we dynamically scale containers for microservices from the Social Network application so as to satisfy SLA. As shown in Fig. 13(a), all schemes could respond to the workload changes promptly. However, Erms can save up to 30% of containers compared to other schemes on average. In Fig. 13(b),




(a) Allocation of containers over time.

(b) P95 end-to-end Latency.

Fig. 13. Performance under the dynamic workload.




(a) Container allocations.

(b) P95 end-to-end Latency.

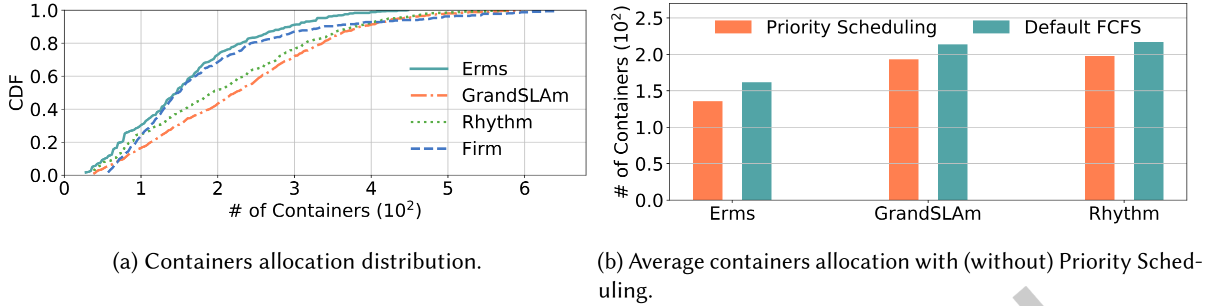
Fig. 14. Performance under dynamic dependency graphs.

we depict the corresponding tail latency of requests submitted over time. It shows that Erms can satisfy SLA requirements all the time without violation, even when the workload grows quickly. However, other schemes can easily violate SLA at peak workloads. In particular, Firm can violate SLA by up to 50% due to its late detection of bottleneck microservices.

*6.3.3 Dynamic dependency graph.* We evaluate the resource allocation and end-to-end latency of services with dynamic dependency graphs using various schemes. To obtain optimal resource allocation for a dynamic graph, we progressively decrease the number of containers for distinct microservices until SLA violations arise. The corresponding resource allocation can then be deemed optimal. To accommodate dynamic dependency graphs, baseline schemes allocate resources for the complete graph rather than its subgraph to prevent SLA violations. We employ a combination of complex and simple graphs to measure the graph's dynamic nature.

As illustrated in Figure 14(a), optimal resource allocation yields an average savings of approximately 5%, 10%, 14%, and 20% compared to Erms, Firm, GrandSLAm, and Rhythm, respectively. These findings demonstrate that Erms outperforms other baseline schemes in dynamic dependency graph scenarios, although a minor gap still exists between Erms and optimal resource allocation. Moreover, the gap between Erms and optimal resource allocation remains stable as the proportion of simple graphs increases, while the gap between other schemes and optimal resource allocation widens with the growth of simple graphs. This is because Erms's accurate modeling of the dependency graph can adapt to the dynamic nature of the graph. Additionally, Figure 14(b) reveals that Erms enhances service performance by 3% compared to other baseline schemes. As the proportion of simple graphs increases, Erms can gradually improve performance due to the benefit of overprovisioning, while the performance of other baseline schemes varies. Consequently, Erms can achieve high performance even under dynamic dependency graph scenarios.

Just Accepted




(a) Containers allocation distribution.

(b) Average containers allocation with (without) Priority Scheduling.

Fig. 15. The benefit brought by individual modules.

### 6.4 Evaluation of Individual Modules

In this subsection, we separately quantify the benefit brought by different components and modules of Erms including Latency Target Computation, Priority Scheduling, and Resource Provisioning.

*6.4.1 Latency target computation.* In this experiment, we evaluate the improvement of Latency Target Computation component by implementing Erms with default FCFS policy to schedule requests at a shared microservice. We compare the overall resource usage across different schemes under various static workloads and SLA settings. The distribution of resource usage is depicted in Fig. 15(a). In an extreme case, Latency Target Computation alone could reduce the overall resource usage by 2× against Firm. On average, Erms outperforms Firm, GrandSLAm and Rhythm by 63.2%, 42.3%, and 61.5%, respectively, indicating that the performance of Erms can degrade a lot without efficient scheduling at shared microservices.

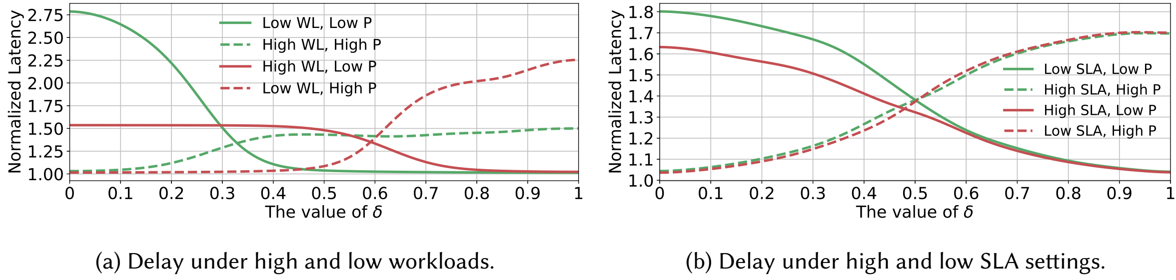
*6.4.2 Benefit of priority scheduling.* We proceed to quantify the benefit brought by Erms' scheduling policy at shared microservices. We also implement priority scheduling under GranSLAm and Rhythm. Firm tunes resource online using a reinforcement learning engine, it is not possible to prioritize requests. Therefore, we only compare Erms to GrandSLAm and Rhythm in this experiment. It is worth noting that priority scheduling requires Erms to recompute latency targets and adjust resource allocation for non-shared microservices as well.

As shown in Fig. 15(b), with priority scheduling, Erms can save about 19% of containers. However, the benefit of priority scheduling for GrandSLAm (Rhythm) is very marginal, i.e., less than 10%. This is because directly applying priority scheduling under GrandSLAm (Rhythm) only reduces resource usage at shared microservices without impacting other microservices. By contrast, Erms relies on priority scheduling to optimize resource allocation for all microservices, leading to increased resource usage for non-shared microservices. However, sacrificing these microservices can benefit shared microservices a lot and therefore greatly reduce the overall resource usage, as illustrated in Fig. 5. This result demonstrates that coordinating latency target computation and scheduling is critical for resource management in shared environments.

We also investigate the impact of the $\delta$ parameter on shared microservices to determine the optimal $\delta$ value under various workload and SLA conditions, as depicted in Fig. 16. For each scenario, we utilize two configurations, with the outcomes represented by green and blue lines in Fig. 16. In the workload scenario, we modify the workload levels of shared microservices for high-priority and low-priority requests.

The green line in Fig. 16(a) reveals that a small $\delta$ value, ranging from 0.05 to 0.1, significantly reduces the latency of low-priority requests under low workloads, while only slightly increasing the latency of high-priority requests under high workloads. Specifically, when $\delta$ is set at 0.1, the latency of low-priority requests decreases by 7.8%, while the latency of high-priority requests increases by a mere 1.3%. Consequently, a $\delta$ value between 0.05 and 0.1 offers high performance for this configuration. As the workload for low-priority requests rises and that for

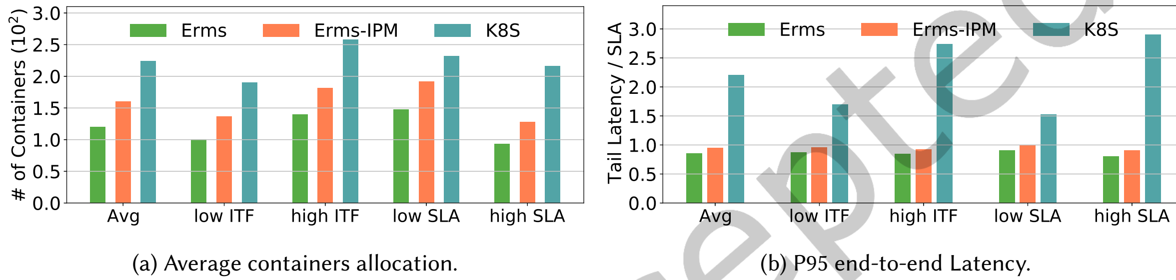
Just Accepted




(a) Delay under high and low workloads.

(b) Delay under high and low SLA settings.

Fig. 16. The delay of requests from services with different priorities (Low P and High P) at shared microservices under various $\delta$.




(a) Average containers allocation.

(b) P95 end-to-end Latency.

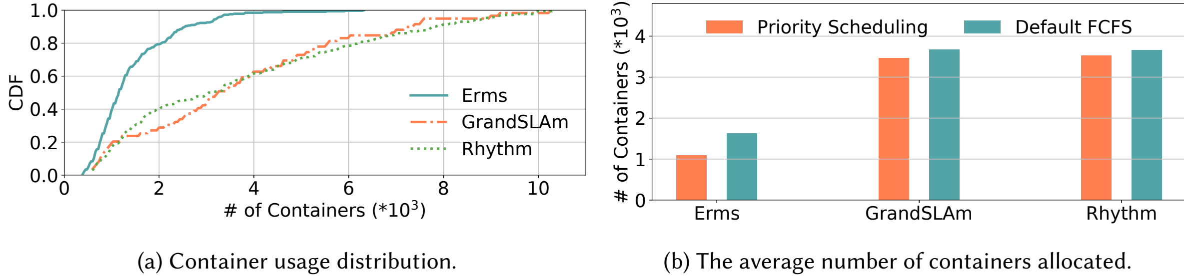
Fig. 17. The benefit of interference (ITF) aware deployment.

Just Accepted

high-priority requests diminishes, as denoted by the red line in Fig. 16(a), the $\delta$ value exhibits minimal influence on the latency of low-priority requests until it surpasses 0.4. This occurs because low-priority services necessitate a higher $\delta$ value to decrease queuing time as their workload increases. A similar observation is evident in distinct SLA scenarios. With a $\delta$ value set between 0.05 and 0.1, the latency of low-priority requests substantially declines, while the latency of high-priority requests experiences a minor increment, as demonstrated in Fig. 16(b).

*6.4.3 Interference-aware containers placement.* In this section, we assess the performance improvement achieved through the implementation of an interference-aware container placement module under the Erms framework (refer to § 5.4). We employ the iBench benchmark [11] to introduce varying degrees of interference, subsequently examining total resource consumption and tail latency under three different approaches: the Erms container placement policy, Erms-IPM, and the default deployment scheme of Kubernetes (K8S).

As illustrated in Figure 17(a), the K8S scheduler necessitates over 50% more containers to fulfill SLA requirements in comparison to Erms-IPM, owing to its lack of resource interference awareness during container placement. Conversely, Erms achieves a 10% reduction in allocated containers relative to Erms-IPM by optimizing end-to-end latency in the presence of resource interference. In high SLA scenarios, the interference-aware container placement module can decrease resource utilization by up to 2×, a more significant effect than in low SLA settings. Two factors contribute to this observed phenomenon. First, high SLA settings result in diminished resource allocation, rendering microservice performance more susceptible to interference from background workloads. Second, high SLA settings lead to high latency targets for each microservice. As microservice latency escalates with interference, resource usage increases to maintain the same latency target under intensified interference. This demonstrates the importance of profiling microservice performance while considering interference-awareness in order to optimize resource allocation effectively.




(a) Container usage distribution.

(b) The average number of containers allocated.

Fig. 18. Simulation results using Alibaba traces.

We further assess the end-to-end latency for services under Erms, Erms-IPM, and K8S while utilizing the same amount of resources. As depicted in Figure 17(b), Erms significantly improves latency performance by 10% and 1.2× on average when compared to Erms-IPM and K8S, respectively. Notably, Erms outperforms K8S by 2.2× in high interference scenarios and by 2× in high SLA settings, showing its enhanced efficiency in optimizing service latency.

### 6.5 Trace-driven Simulations

To evaluate Erms on a large scale, we replay Alibaba microservice workloads to conduct trace-driven simulations for Taobao Application. This application includes 500+ services and each service contains 50 microservices on average. The total number of shared microservices is 300+.

*6.5.1 End-to-End performance.* We depict the distribution of the total number of containers deployed under each service in Fig. 18(a). It shows that more than 80% of services require less than 2000 containers under Erms, whereas these services need 6000 containers under both GrandSLAm and Rhythm. In addition, Erms could reduce the number of allocated containers by 1.6× on average, compared to baseline schemes, as shown in Fig. 18(b). This improvement is much larger than that under real benchmarks, demonstrating Erms has more opportunities to improve resource efficiency for services with complex call dependency. We also evaluate the improvement of Latency Target Computation and Priority Scheduling, respectively. Results in Fig. 18(b) show that Latency Target Computation alone can save resource usage by up to 1.2×. By contrast, Priority Scheduling leads to a reduction in resource usage by 50%. This improvement is also much higher than that from benchmarks since there are more shared microservices in Alibaba traces.

*6.5.2 Scalability of Erms.* We evaluate the scaling overhead of Erms using Alibaba traces since their scale is much larger than that of DeathStarBench. The average overhead of Latency Target Computation is 15ms on an Intel Xeon CPU. For the largest graph with 1000+ microservices, the computational overhead is 300ms. In addition, the overhead of resource provisioning is 200ms on average. Most of time, Erms only needs to scale no more than 1000 containers across 5000 hosts. Therefore, the overall scaling overhead is quite small since a container usually requires several seconds to start [38].

## 7 DISCUSSION

In this section, we will discuss several practical issues about deploying Erms in a production environment.

*Modelling latency using linear functions.* Erms chooses to quantify microservice latency using piece-wise linear functions. The key reason is that these functions can well model microservice behavior, as explained in § 2.2. Moreover, the piece-wise linear function can achieve up to 86% profiling accuracy on Alibaba production workloads and DeathStarBench, even outperforming complicated models including XGBoost and Neural Network. Another advantage is that Erms can leverage piecewise linear functions to derive closed-form expressions that

Just Accepted

assign optimal latency targets to each microservice. As a result, Erms can achieve better performance than existing heuristics while being scalable to handle large-scale problems. In fact, linear functions are not satisfactory for only very few microservices, i.e. less than 3% in DeathStarBench with profiling accuracy around 62%. This is because the latency of these microservices is relatively small, making it difficult to predict accurately. Nonetheless, these microservices have a negligible impact on the end-to-end SLA, and Erms only allocates a small amount of resources to them.

*Handling resource-related exceptions.* Resource-related exceptions, such as out of memory, rarely happen under Erms for two reasons. First, Erms computes latency targets across microservices based on SLA requirements and current workload. Erms assigns each microservice a proper number of containers based on the latency target to avoid overload. Second, Erms rounds up the number of containers per microservice to an integer. In this sense, Erms can eliminate the negative impact of mispredictions to avoid exceptions. Also, this over-provisioning due to rounding up is negligible relative to the total number of containers per microservice (typically hundreds to thousands in production environments).

## 8 RELATED WORK

**Microservice autoscaling**. GrandSLAm builds an execution framework for ML-based microservices [23]. However, it allocates microservice latency targets independently among different services without global coordination. Microscaler [45] adopts Bayesian optimization approach to scale the number of instances for those important microservices. Rhythm [47] builds an advanced model to quantify the contribution of each microservice. Firm [38] leverages machine-learning techniques to localize critical microservice that can have a heavy impact on the overall service performance under low-level resource interference. Most recently, Sinan [46] presents a CNN-based cluster resource manager for microservice architecture to guarantee QoS while maintaining high resource utilization. DeepRest [9] and Graf [36] employ graph neural networks to accurately estimate resource allocation in microservices, particularly those with intricate dependency graphs. Meanwhile, ORION [31] models serverless latency as a stochastic distribution and subsequently utilizes convolution operations to determine the end-to-end latency for serverless applications. LAOrchestrator [35] designs a double nested learning algorithm to dynamically provision the number of containers for ad-hoc data analytics. ATOM [21] and MIRAS [44] tunes resources for microservices to improve the overall system throughput. All of these works do not investigate shared microservices.

**Microservice sharing**. To handle microservice sharing, Q-Zilla [34] designs a decoupled size-interval task scheduling policy to minimize microservice tail latency based on resource reservation. $\mu$steal [33] partitions resources at shared microservice and makes use of stealing to improve utilization. However, these schemes are not suitable for practical microservice architecture since they need to know the processing time of each microservice call in advance. Moreover, optimizing individual microservice latency can not provide SLA guarantees on the end-to-end performance of online services.

**Graph analysis**. Sage [17] builds a graphical model to identify the root cause of unpredictable microservice performance and dynamically adjust resources accordingly. This is not scalable in a production environment since a practical application can even consist of hundreds of microservice with complicated parallel or sequential dependencies. Parslo [32] adopts a gradient descent-based approach to break the end-to-end SLA into small unit SLO. However, such an iterative approach is generally costly in time, and can not be applicable to dynamic workloads. Llama [40] and Kraken [5] aim to optimize performance for serverless systems, which can not be applied to general microservices.

**Interference mitigation:** The problem of resource interference in cloud-related systems has been extensively investigated in the literature [7, 12, 27, 37]. These works focus on the co-scheduling of different applications, aiming at maximizing application performance. The intention of Erms is different from these works, Erms aims to

Just Accepted

minimize resource unbalance across different hosts so as to improve resource efficiency and provide end-to-end performance guarantees.

## 9 CONCLUSION

This paper presents a new method for dynamically allocating resources in shared microservice architectures through the use of explicit modeling. Our designs incorporate prioritization among various services, providing valuable insights into the effective deployment of online services. However, one limitation of Erms is its tendency to overprovision resources for online services with highly dynamic dependency graphs, as demonstrated in our experiments. A more promising approach would involve estimating resource allocation for graphs exhibiting different levels of dynamics, rather than relying solely on a complete graph. This would enable the scaling of minimal resources to satisfy the SLA for online services with diverse dependency graphs.

## ACKNOWLEDGMENTS

This work was supported in part by Guangdong Key-Area Research and Development Program (NO.2020B010164003), the National Natural Science Foundation of China (No. 62072451, 92267105), the Science and Technology Development Fund of Macau (0024/2022/A1), Guangdong Special Support Plan (No. 2021TQ06X990), and Alibaba Innovative Research Program.

Just Accepted

## REFERENCES


[1] 2021. Alibaba Microservices Cluster Traces. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021.
[2] 2022. Jaeger. https://jaegertracing.io/.
[3] 2022. Prometheus. https://prometheus.io/.
[4] Azure Cloud Container Apps. 2022. https://azure.microsoft.com/en-us/services/container-apps/.
[5] 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 *Proceedings of SoCC.*
[6] S. Boyd and L. Vandenberghe. 2004. *Convex Optimization.* Cambridge University Press, Chapter 5.
[7] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of ASPLOS.*
[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of SIGKDD.*
[9] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. 2022. DeepRest: Deep Resource Estimation for Interactive Microservices. In *Proceedings of EuroSys.*
[10] Docker containers. 2022. https://www.docker.com/.
[11] Christina Delimitrou and Christos Kozyrakis. 2013. IBench: Quantifying interference for datacenter applications. In *Proceedings of IISWC.*
[12] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of ASPLOS.*
[13] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, and Manuel Mazzara et al. 2018. Microservices: How To Make Your Application Scale. In *Lecture Notes in Computer Science.*
[14] Alibaba Cloud Microservices Engine. 2022. https://www.alibabacloud.com/product/microservices-engine.
[15] Google Kubernetes Engine. 2022. https://cloud.google.com/kubernetes-engine.
[16] Susan Fowler. 2016. *Production-ready Microservices: Building Standardized Systems Across an Engineering Organization.* O'Reilly Media.
[17] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical & Scalable ML-Driven Performance Debugging in Microservices. In *Proceedings of ASPLOS.*
[18] Yu Gan, Yanqi Zhang, et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of ASPLOS.*
[19] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of ASPLOS.*
[20] Anshul Gandhi and Amoghvarsha Suresh. 2019. Leveraging Queueing Theory and OS Profiling to Reduce Application Latency. In *International Middleware Conference Tutorials.*
[21] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-Driven Autoscaling for Microservices. In *Proceedings of ICDCS.*

[22] Mingyi Hong and Zhi Quan Luo. 2016. On the linear convergence of the alternating direction method of multipliers. *Mathematical Programming* 162 (2016).

[23] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, and Jason Mars. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of Eurosys*.

[24] Kubernetes. 2022. https://kubernetes.io..

[25] Mingyu Liang, Yu Gan, Yueying Li, Carlos Torres, Abhishek Dhanotia, Mahesh Ketkar, and Christina Delimitrou. 2023. Ditto: End-to-End Application Cloning for Networked Cloud Services. In *Proceedings of ASPLOS*.

[26] Qixiao Liu and Zhibin Yu. 2018. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba trace. In *Proceedings of ACM SoCC*.

[27] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of ISCA*.

[28] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of ACM SoCC*.

[29] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Cheng-Zhong Xu. 2022. An In-depth Study of Microservice Call Graph and Runtime Performance. *IEEE Transactions on Parallel and Distributed Systems*.

[30] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 2023. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees. In *Proceedings of ASPLOS*.

[31] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *Proceedings of OSDI*.

[32] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F Wenisch. 2021. Parslo: A Gradient Descent-based Approach for Near-optimal Partial SLO Allotment in Microservices. In *Proceedings of ACM SoCC*.

[33] Amirhossein Mirhosseini and Thomas F. Wenisch. 2021. $\mu$Steal: A Theory-backed Framework for Preemptive Work and Resource Stealing in Mixed-Criticality Microservices. In *Proceedings of ICS*.

[34] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. 2020. Q-Zilla: A Scheduling Framework and Core Microarchitecture for Tail-tolerant Microservices. In *Proceedings of HPCA*.

[35] Jennifer Ortiz, Brendan Lee, Magdalena Balazinska, Johannes Gehrke, and Joseph L. Hellerstein. 2018. SLAOrchestrator: Reducing the Cost of Performance SLAs for Cloud Data Analytics. In *Proceedings of ATC*.

[36] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: A Graph Neural Network based Proactive Resource Allocation Framework for SLO-Oriented Microservices. In *Proceedings of ACM CoNext*.

[37] Tirthak Patel and Devesh Tiwari. 2020. CLITE: Efficient and QoS-Aware Co-location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *Proceedings of HPCA*.

[38] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *Proceedings of OSDI*.

[39] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[40] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. *Proceedings of ACM SoCC*.

[41] Krzysztof Rzadca, Pawel Findeisen, et al. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of EuroSys*.

[42] Akshitha Sriraman and Thomas F Wenisch. 2018. $\mu$Tune: Auto-Tuned Threading for $OLDI$ Microservices. In *Proceedings of OSDI*.

[43] Microservices workshop. 2022. http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference/.

[44] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. 2019. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *Proceedings of ICDCS*.

[45] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic Scaling for Microservices with an Online Learning Approach. In *Proceedings of ICWS*.

[46] 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 *Proceedings of ASPLOS*.

[47] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of EuroSys*.

[48] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of ACM SoCC*.

[49] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Poster: Benchmarking microservice systems for software engineering research. In *Proceedings of ICSE*.


Just Accepted