Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees

Shutian Luo*‡
Shenzhen Institute of Advanced Technology, CAS
Univ. of CAS, Univ. of Macau
Macau SAR, China
st.luo@siat.ac.cn

Huanle Xu*‡
University of Macau
Macau SAR, China
huanlexu@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

Jian He
Alibaba Group
Hangzhou, China
jian.h@alibaba-inc.com

Guodong Yang
Alibaba Group
Hangzhou, China
luren.ygd@taobao.com

Chengzhong Xu†‡
University of Macau
Macau SAR, China
czxu@um.edu.mo

ABSTRACT

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 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.

*Co-first author. Both authors contributed equally to this paper.
†Corresponding author.
‡S. Luo, H. Xu, K. Ye and C. Xu are also with Guangdong-Hong Kong-Macao Joint Laboratory of Human-Machine Intelligence-Synergy Systems.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**.

KEYWORDS

Shared Microservices, Resource Management, SLA Guarantees

ACM Reference Format:
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 the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3567955.3567964

1 INTRODUCTION

Recent years have witnessed a rapid emergence and wide adoption of microservice architecture in cloud data centers [4, 13, 14]. 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, 39, 41, 46]. 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 [12, 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, 26]. 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 [9] are usually colocated with batching

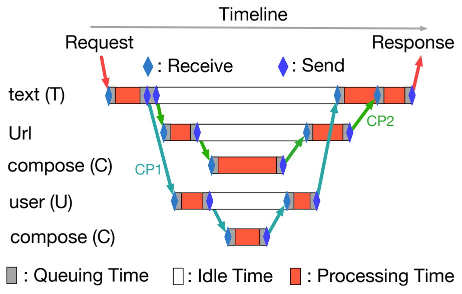


Figure 1: A microservice *dependency graph* where microservice T calls *downstream* microservice Url and U in parallel and calls Url and C sequentially. The end-to-end latency is the time duration between T receiving the request and returning the result.

applications [24]. Resource interference can cause performance imbalances between containers from the same microservice, especially when the workload is heavy. Third, a single microservice can be shared among multiple services [26, 40]. These services can present diverse workload patterns and have different SLA requirements.

Existing approaches provide SLA guarantees for microservice management via handcrafted heuristics, reinforcement learning approaches, or deep learning algorithms [7, 22, 33, 35, 38, 44, 45]. 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 [22, 45]. 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 [35]. 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 [33, 44]. However, this is not scalable for complex services in production environments where a service can consist of 1000+ microservices with many tiers [26].

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 presenting Erms for Efficient Resource Management in a Shared Microservice Execution Framework with SLA Guarantees. 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 [33, 44] and other heuristic solutions [22, 45].

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 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 [23]. We evaluate Erms via real deployment as well as large-scale trace-driven simulations. 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 [23], 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 [27]. 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) to serve all requests sent to it.

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 [35]. 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

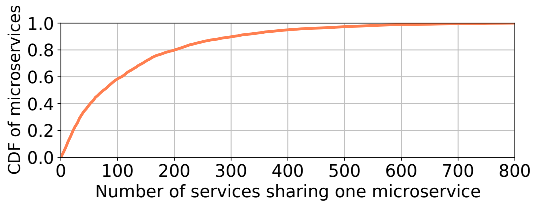


Figure 2: The cumulative distribution of microservices shared by a different number of online services from Alibaba traces.

$CP_2 = \{T, Url, C\}$. 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

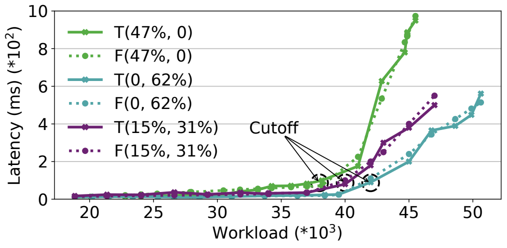
Compared to the end-to-end latency of online service, microservice latency is treated as a more fine-grained metric in terms of quantifying the resource pressure of deployed containers. Due to this, recent works begin to investigate how this performance metric can be affected by various factors such as the workload and resource interference on the physical host [7, 22, 45].

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, 46]. 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]. Stem from 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 ith 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:

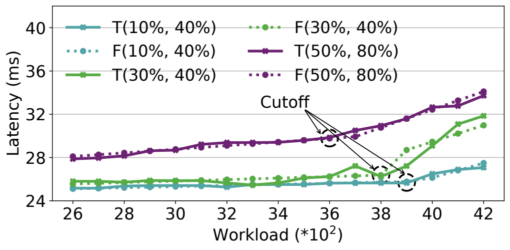
$$L_i^m = (S_i^m - R_i^m) - (S_i^d - R_i^d). \quad (1)$$

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.

The study of microservice latency in existing works focuses on the first and second-order statistics across different workloads.



(a) DeathStarBench [18].



(b) Alibaba traces [1].

Figure 3: P95 Microservice latency from different traces (P99 behaves similarly). The two numbers in each bracket represent the host CPU and memory utilization. T is for ground truth and F is for fitting using a piece-wise linear function.

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, there is a cut-off point (marked with a black circle) below which the tail latency increases slowly and almost linearly in the workload. By contrast, when the workload exceeds this point, microservice latency increases much faster (almost linearly) with the increase of workload. 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 [22, 45].

Another limitation is that, existing studies do not quantify the impact of resource interference on the slope of the latency curve [22, 44, 45]. Here, we measure resource interference in terms of CPU and memory resource usage on physical hosts. Our quantification of microservice latency shows that the slope changes when interference varies. As shown in Fig. 3(a), on a host with 47% CPU utilization, the rate of increase in microservice latency after the cutoff is 5 times that on a host with 62% memory utilization. Moreover, resource interference forces the cut-off point to move forward. Specifically, when interference becomes more serious, microservice latency starts to increase fast earlier.

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

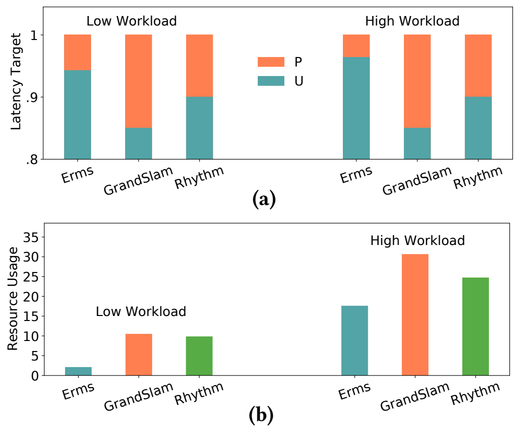


Figure 4: An online service calls userTimeline (U) and postStorage (P) sequentially. The latency of Microservice U is more sensitive to workload changes than that of P. (a) Computed latency targets under different schemes in low-workload and high-workload settings. (b) The normalized total resource usage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

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.

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 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 [22] and Rhythm [45]. 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 graphs* 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

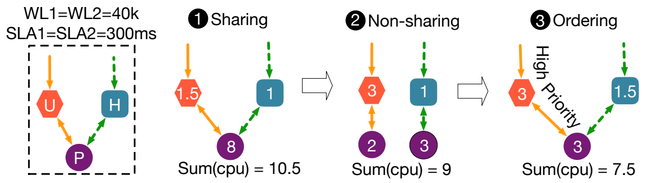


Figure 5: Resource usage under microservice multiplexing for ensuring SLA requirement, the number in each circle represents the amount of CPU allocation to a microservice.

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.

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.

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 Jaeger [2]. *Tracing Coordinator* generates microservice dependency graphs and extracts the individual microservice latency based on historic traces.

Erms includes an *Offline Profiling* module (❷ in Fig. 6) that works in the background. It fetches all microservice latency samples under different workloads for all deployed containers for each microservice from the *Tracing Coordinator*. With these data samples, the offline module builds a fitting model that profiles microservice tail latency as a piece-wise linear function of the workload. Parameters of these functions also vary with resource interference on the physical host.

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 (❸ in Fig. 6), Latency Target Computation (❹ in Fig. 6), and Priority Scheduling (❺ 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.

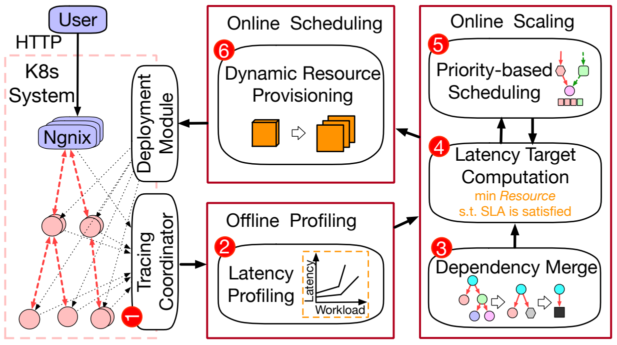


Figure 6: The system architecture of Erms.

Erms also contains a *Resource Provisioning* module (❻ in Fig. 6) to place all containers from different microservices across physical hosts in the cluster. Whenever *Online Scaling* module determines to scale down or scale out, *Resource Provisioning* module chooses to place newly scheduled containers or release existing ones such that the overall resource interference is minimized. 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).

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:

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

$\vec{n} = \langle n_1, n_2, \cdots, n_N \rangle$ 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. $\text{latency}_k(\vec{n})$ and SLA_k represent the tail end-to-end latency of requests from service k under resource allocation $\vec{n}$ and the SLA requirement of service k, respectively.

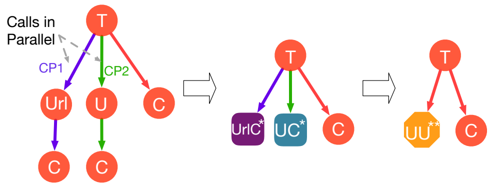


Figure 7: Erms simplifies the structure of a general graph via gradually removing parallel dependency.

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 γ_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, latency$_k(\vec{n})$ can be formulated as:

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

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 γ_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 latency$_k(\vec{n})$. 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 γ. 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.

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).

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 [26]. (Line 7 to Line 19). Each two-tier invocation consists

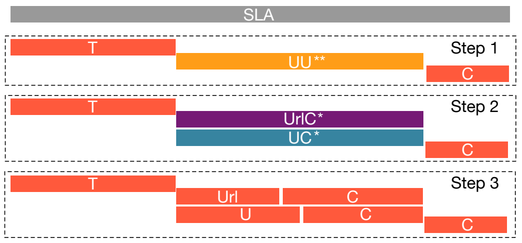


Figure 8: An example of computing latency target for microservice graph in Fig. 7.

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.

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
```

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 γ_1, γ_2 and SLA requirements SLA$_1$ and 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:

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

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

where Φ_1 and Φ_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 γ_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 $\overrightarrow{n}$.

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. More details about the proof of the theorem can be found in appendix A.

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.*

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 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

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. In addition, microservices such as e-commerce and web services typically process light-weight inputs that consume only a small amount of data; therefore, input data sizes of requests are negligible and have little impact on microservice latencies. As such, Erms only takes into account workload and resource interference when profiling microservice latency.

In terms of resource interference, Erms mainly considers CPU utilization and memory utilization of the physical host where the microservice container is located. The reason why memory utilization is critical to quantifying interference is that high memory utilization is prone to trigger memory compaction in an operating system, thereby significantly degrading the performance of processes [47]. Specifically, memory pages are usually scattered among processes, and when memory utilization is high, memory fragmentation problems can easily occur. Consequently, the operating system needs to perform memory compaction when allocating physically-continuous pages for a process, which will suspend the process execution. Additionally, we have compared the interference effect of memory bandwidth and memory utilization by measuring microservice latency at the same utilization level of these two metrics using DeathStarBench. The result shows that for quantifying interference, memory utilization is almost as important as memory bandwidth.

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 jth 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 jth minute, i.e., γ_i^j. These two together with the average resource utilization are regarded as one data sample for Microservice i, i.e., $d_i^j = (L_i^j, \gamma_i^j, \mathrm{C}_i^j, \mathrm{M}_i^j)$ where the last two elements represent CPU and memory utilization respectively. Erms fits all these samples into a piece-wise model as shown below.

$$L_i^j = \begin{cases} (\alpha_i^1 \mathrm{C}_i^j + \beta_i^1 \mathrm{M}_i^j + c_i^1)\gamma_i^j + b_i^1, & \gamma_i^j \le \sigma_i, \\ (\alpha_i^2 \mathrm{C}_i^j + \beta_i^2 \mathrm{M}_i^j + c_i^2)\gamma_i^j + b_i^2, & \text{otherwise.} \end{cases} \tag{15}$$

Given resource interference, i.e., fixing C_i^j and M_i^j, L_i^j is a piece-wise linear function of the workload γ_i^j. $(\alpha_i^l, \beta_i^l, c_i^l, b_i^l)_{l=1,2}$ are fixed parameters learned directly from historical traces. In contrast, the cut-off point σ_i is a function of resource interference; Erms applies the decision tree model [36] to learn it.

The profiling model of Erms described above can be easily extended to include various shared resources, including memory bandwidth, LLC, and network bandwidth. However, taking all these factors into account can result in large profiling overhead. In fact, CPU and memory utilization alone are sufficient for achieving good profiling performance, as demonstrated in § 6.2.

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 σ_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 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,

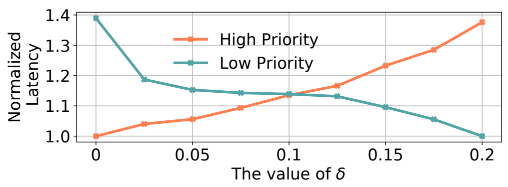


Figure 9: Response time of requests under various δ.

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 lth highest priority are scheduled with probability $\delta^{l-1}(1-\delta)$, and the service with the lowest priority is scheduled with probability δ^{n-1} where n is the number of services. Here, a small δ 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. To investigate the impact of δ on shared microservice, we run a simple experiment where two representative services share a microservice under various workloads. We examine all shared microservices from DeathStarBench. In most cases, the value of δ has a minor effect on the response time of both high-priority and low-priority requests. In the worst case, increasing δ from 0 to 0.05 only degrades tail performance (95th percentile) of high-priority requests by 5%, whereas it can improve the performance of low-priority requests by more than 20%, as shown in Fig. 9. According to this observation, Erms set δ to 0.05.

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 graphs* behave like a tree [26], 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 Resource Provisioning

For scalability, *Online Scaling* module only feeds the average resource interference across different hosts into microservice profiling model. However, scheduled containers of a microservice can be deployed across various hosts, which may lead to different levels of resource interference. This can result in unbalanced performance of containers from the same microservice, causing SLA violations. To mitigate this effect, *Resource Provisioning* module chooses hosts to place or release containers that can minimize resource unbalance across all hosts. For each host, the resource unbalance is defined as the difference between the host utilization and the cluster-wide utilization. When computing the host utilization, Erms sums up the actual resource usage of all containers deployed on that host. By contrast, cluster-wide utilization is the average host utilization in a cluster.

This provisioning problem turns out to be a non-linear integer programming problem, which is NP-hard. When the number of hosts and deployed containers is large, solving the optimization problem can be time-consuming, thereby limiting its applicability to a production environment. To mitigate this overhead, Erms statically divides the hosts of a cluster into multiple groups of the same size and solves a much smaller scale optimization problem, following the recently developed POP optimization technique [31].

5.5 Erms Implementation

We implement a prototype of Erms on top of Kubernetes [23], 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.

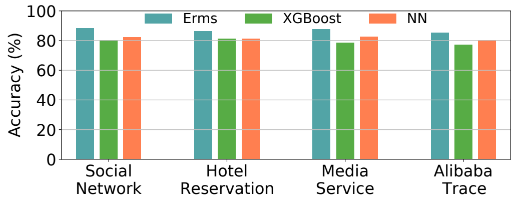
6 EVALUATION OF ERMS

6.1 Experiment Setup

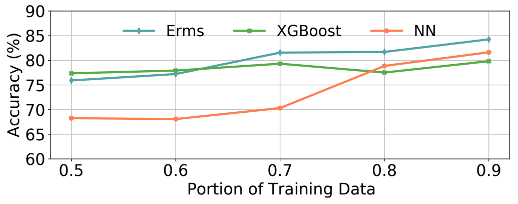
Benchmarks: We evaluate Erms using an open-sourced microservice benchmark, DeathStarBench [18], consisting of Social Network, Media Service, and Hotel Reservation applications. These applications contain 36, 38, and 15 unique microservices respectively, and include 3, 1, and 4 different services. Moreover, both Social Network application and Hotel Reservation application have 3 shared microservices.

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 200M 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 [26]. 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.

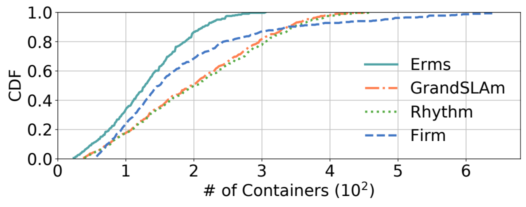


(a) Profiling accuracy under different APP and Alibaba traces.

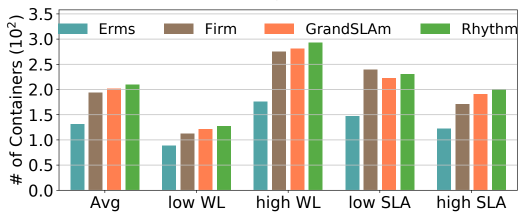


(b) Profiling accuracy using different portions of training data.

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



(a) Resource usage distribution.



(b) Average resource usage.

Figure 11: Containers allocated with static workloads.

Baseline Schemes: We compare Erms against GrandSLAm [22] and Rhythm [45]. In addition, we also implement Firm [35] as one additional baseline scheme.

- **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.

6.2 Microservice Profiling Accuracy

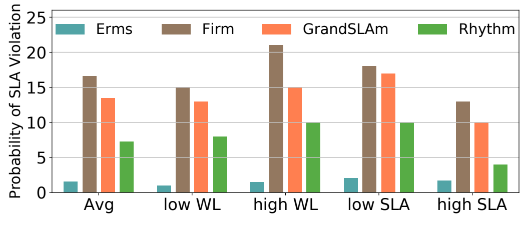
To validate the accuracy of Erms' microservice profiling module, we run DeathStarBench in our local cluster and collect one-day running samples for each microservice. We fix the interference level on each host via injecting iBench workloads [10] during each hour, and collect one sample per minute for 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 [26]. 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 88% 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 81% 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.

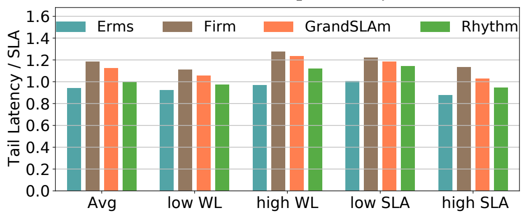
6.3 Resource Efficiency and Performance

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 80% of workloads require less than 200 containers under Erms, while these workloads need about 300 containers under both GrandSLAm, and Rhythm. GrandSLAm and Rhythm have similar distributions of resource usage as 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 48.1%, 53.5% and 60.1% 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

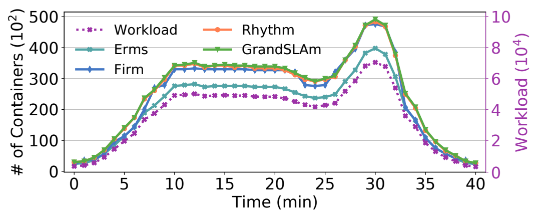


(a) SLA violation probability.

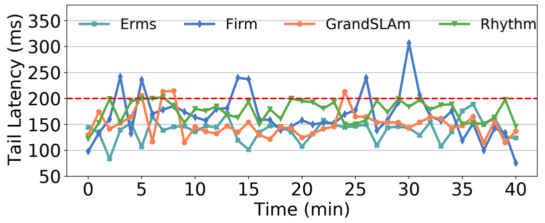


(b) P95 end-to-end Latency.

Figure 12: Tail latency under different schemes.



(a) Allocation of containers over time.



(b) P95 end-to-end Latency.

Figure 13: Performance under the dynamic workload.

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 2%, whereas it is as high as 16.5%, 13.5%, and 7.3% 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 10% 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), 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.

6.4 Evaluation of Individual Modules

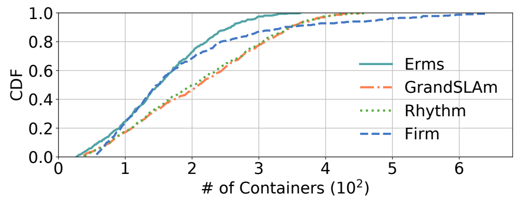
In this subsection, we separately quantify the benefit brought by different components and modules of Erms including Latency Target Computation, Priority Scheduling, and Resource Provisioning.

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. 14(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 19%, 35.8%, and 33.4%, respectively, indicating that the performance of Erms can degrade a lot without efficient scheduling at shared microservices.

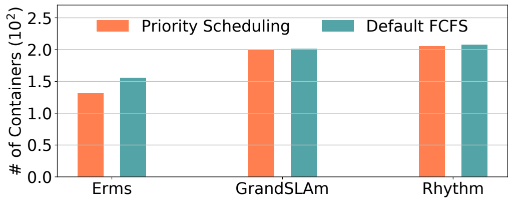
6.4.2 Benefit of priority scheduling. We proceed to quantify the benefit brought by Erms' scheduling policy at shared microservices. We also implement priority scheduling under GranSLAm and Rhythm. Firm tunes resource online using a reinforcement learning engine, it is not possible to prioritize requests. 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. 14(b), with priority scheduling, Erms can save about 20% of containers. However, the benefit of priority scheduling for GrandSLAm (Rhythm) is very marginal, i.e., less than 5%. 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.

6.4.3 Interference-aware provisioning. In this part, we aim to evaluate the performance gain of interference-aware provisioning module under Erms (§ 5.4). To achieve this, we use iBench workload to inject different levels of interference [10]. We quantify the total resource usage and tail latency under both Erms's provisioning policy and K8S' default deployment scheme. As shown in Fig. 15(a), K8S requires more than 50% of containers to satisfy SLA requirements since it is unaware of resource interference. When referring

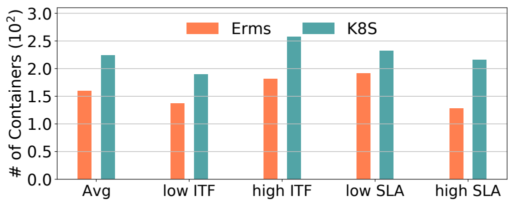


(a) Containers allocation distribution.

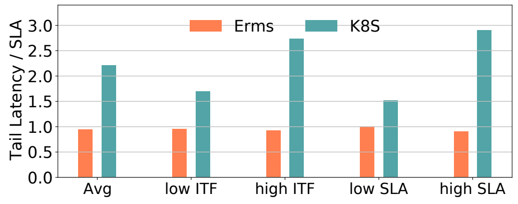


(b) Average containers allocation with (without) Priority Scheduling.

Figure 14: The benefit brought by individual modules.



(a) Average containers allocation.



(b) P95 end-to-end Latency.

Figure 15: The benefit of interference (ITF) aware deployment.

to the high SLA setting, the interference-aware provisioning module can reduce resource usage by 2×, which is more significant than that under the low SLA setting. There are two reasons behind this phenomenon. First, high SLA leads to less resource allocation, making microservice containers easily impacted by the background workload. Second, high SLA results in a high latency target for each microservice. Because microservice latency increases with interference (shown in Fig. 3), the amount of resource usage grows when interference increases under the same latency target. This demonstrates profiling microservice performance with interference-awareness is crucial for saving resources.

We also estimate the end-to-end latency for services under Erms and K8S when they use the same amount of resources. Fig. 15(b) shows that Erms could improve latency performance by 1.2× on average. Improvement of Erms reaches 2.2× and 2× in high interference and high SLA scenarios, respectively.

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. 16(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. 16(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. 16(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.

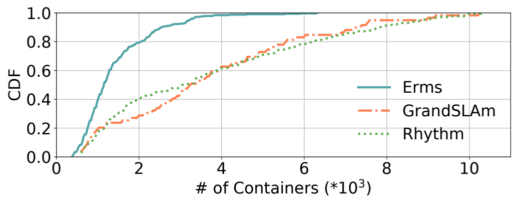
6.5.2 Scalability of Erms. We evaluate the scaling overhead of Erms using Alibaba traces since their scale is much larger than that of DeathStarBench. The average overhead of Latency Target Computation is 15ms on an Intel Xeon CPU. For the largest graph with 1000+ microservices, the computational overhead 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 [35].

7 DISCUSSION

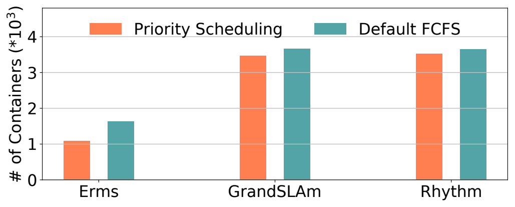
In this section, we will discuss several practical issues about deploying Erms in a production environment.

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 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 dynamic dependencies. Erms mainly focuses on static microservice dependency graphs, which do not change with different input data of user requests. However, the graphs are not necessarily static and can vary significantly in highly dynamic execution



(a) Container usage distribution.



(b) The average number of containers allocated.

Figure 16: Simulation results using Alibaba traces.

environment [26]. For those dynamic graphs generated from the same online service, Erms compares their differences and merges them into a complete dependency graph. Erms then allocates resources based on the complete graph to satisfy SLA requirements. For highly dynamic graphs, Erms tends to overprovision resources because a request is usually handled by a small set of microservices in the complete graph. Optimizing this problem will be our future work.

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 [22]. However, it allocates microservice latency targets independently among different services without global coordination. Microscaler [43] adopts Bayesian optimization approach to scale the number of instances for those important microservices. Rhythm [45] builds an advanced model to quantify the contribution of each microservice. Firm [35] 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 [44] presents a CNN-based cluster resource manager for microservice architecture to guarantee QoS while maintaining high resource utilization. LAOrchestrator [32] designs a double nested learning algorithm to dynamically provision the number of containers for ad-hoc data analytics. ATOM [21] and MIRAS [42] 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 [30] designs a decoupled size-interval task scheduling policy to minimize microservice tail latency based on resource reservation. μsteal [29] 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 [28] 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 [37] 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, 11, 25, 34]. 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 minimize resource unbalance across different hosts so as to improve resource efficiency and provide end-to-end performance guarantees.

9 CONCLUSION

This paper is the first attempt to dynamically provision resources for a shared microservice architecture through explicit modeling. Our designs such as enforcing prioritization among different services provide further insights on efficient deployment of online services. One limitation of Erms is that it tends to overprovision resources for online services with highly dynamic dependency graphs. A more advanced solution is to cluster graphs into multiple classes and scale resources in each class instead of a complete graph. Furthermore, generalizing our interference-awareness model to include more types of shared resources with scalable solutions will be explored in our future work as well.

10 DATA-AVAILABILITY STATEMENT

The source codes of Erms system is available in Zenodo [15].

ACKNOWLEDGEMENTS

We sincerely thank the anonymous ASPLOS'23 reviewers for their insightful suggestions and Chenyu Lin for his significant help on the release of Erms system. This work is supported in part by Key-Area Research and Development Program of Guangdong Province under Grant No.2020B010164003, the National Natural Science Foundation of China (62072451), the Start-up Research Grant of University of Macau (SRG2021-00004-FST), and Alibaba Group through Alibaba Innovative Research Program.

A THEORETICAL FOUNDATIONS

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, SLA_1 can be formulated as:

$$a_u \frac{\gamma_1}{n_u} + b_u + a_p \frac{\gamma_1 + \gamma_2}{n_p} + b_p \leq \mathrm{SLA}_1, \tag{16}$$

and 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:

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

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:

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

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 close-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:

$$\begin{aligned} RU^o \leq & \frac{\left(\sqrt{a_h\gamma_2 R_h} + \sqrt{a_p(\gamma_1+\gamma_2)R_p}\right)^2}{\mathrm{SLA}_1 - b_u - b_p} + a_u\gamma_1 R_u \\ & + \sqrt{a_u a_p R_u R_p}\gamma_1. \end{aligned} \tag{19}$$

Moreover, it can be readily shown that the R.H.S. of Eq. (19) 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.

B ARTIFACT APPENDIX

B.1 Abstract

Erms is a cluster-level resource management system for shared microservices with SLA guarantees. This artifact includes a prototype implementation of Erms and the experimental workflows for running DeathStarBench on Erms.

B.2 Artifact Check-list (meta-information)

- **Data set:** Erms' Tracing Coordinator adopts Prometheus and Jaeger to record the runtime metrics, including both OS-level metrics and application-level metrics. In addition, we use Alibaba microservice traces to conduct large-scale trace-driven simulations.
- **Run-time environment:** Erms is deployed on top of Kubernetes cluster.
- **Hardware:** Experiments are conducted on a local cluster of 20 two-socket physical hosts. Each host is configured with 32 CPU cores and 64 GB RAM.
- **Metrics:** We adopt the resource usage (i.e., the number of allocated containers) and the end-to-end latency of online services as performance metrics to evaluate Erms.
- **Output:** The final output is the resource scaling policy for each microservices under the given SLA requirement.
- **Experiments:** This artifact includes all the scripts and benchmarks needed to evaluate the functionality and reproduce the results of the paper.
- **How much time is needed to complete experiments (approximately)?** It takes nearly one week to conduct the complete experiments. In particular, offline profiling alone needs three days to profile an online service consisting of multiple microservices. The artifact also provides an estimation of the execution time taken for running each step.
- **Publicly available?** Yes. User can get access to the source code of Erms through GitHub: https://github.com/Cloud-and-Distributed-Systems/Erms.
- **Archived (provide DOI)?**: Yes. We archived the source code of Erms in Zenodo with DOI: https://doi.org/10.5281/zenodo.7220659.

B.3 Description

B.3.1 How to access. Erms's source code is available on GitHub: https://github.com/Cloud-and-Distributed-Systems/Erms.

B.3.2 Hardware dependencies. Our cluster consists of 20 two-socket physical hosts with x86-64 architecture. Each host is configured with 32 CPU cores and 64 GB memory. All CPU cores are Intel Xeon E5-2630. The type of host memory is DDR4.

B.3.3 Software dependencies. Kubernetes version is v1.18. Docker Version is 20.10.4. Jeager version is v1.15.0 and Prometheus version is v2.16.

B.3.4 Data sets. The scripts to generate the datasets used for offline profiling are included in the artifact. The README file describes the usage of the scripts. In addition, the Alibaba microservice traces for the trace-driven simulation are vailable at [26].

B.4 Installation

The artifact includes how to deploy Kubernetes cluster on physical machines and install the dependencies of Erms. We have prepared the required Python packages. Please run the script to install the packages as follows.

```
$ pip3 install -r requirements.txt
```

B.5 Experiment Workflow

Erms is deployed on top of a Kubernetes cluster. First, Erms runs DeathStarBench applications at different levels of interference and different workloads, and collects the generated traces. Erms then builds the performance model for each microservice based on these traces. Next, Erms computes a latency target for each microservice with given SLAs and workloads, and performs priority scheduling on shared microservices. At the same time, Erms determines the number of containers for each microservice according to its performance model and the computed latency target. Finally, to mitigate the impact of resource interference, Erms selects appropriate hosts to place or release containers to balance resource utilization among hosts. Please check the Github README for more details.

B.6 Functional Evaluation for Erms

In this part, we evaluate the function of Erms' each module separately and the evaluated result will be printed in the terminal. Please refer to § 3 for the detail of each module.

B.6.1 Tips for artifact evaluation. We provide some tips for users to run Erms easily.

- Before executing a script, please check the configuration and help messages of the script. Configuration files can be found in *AE/scripts/configs/${module_name}.yaml*. Users can check the usage, explanation and arguments of the script via running *./script.sh -h*.
- Since some scripts take a long time to run, please build a screen via *screen -S session_name*, to maintain a long-live ssh connection.
- If it fails when running the script, user *should* delete the result directory if exist and run the script again.

B.6.2 Offline profiling. Erms use pair-wised linear functions to profile microservices performance. The profiling process takes more than two days via the script with default settings as follows.

```
$ ./AE/scripts/profiling-testing.sh
```

The profiling data can be found in *AE/data/data_{app}/*. Users can tune the interference configurations and the repeat times to balance the profiling overhead and the accuracy. Run *./AE/scripts/profiling-testing.sh -h* to refer more details about the interference configurations.

Users can profile the performance of microservices with the collected data via the script.

```
$ ./AE/scripts/profiling-fitting.sh
```

B.6.3 Online scaling. Online Scaling determines how many containers are allocated to each microservices, which includes three parts, i.e., dependency merge, latency target computation and priority-based scheduling. Users can evaluate the three parts separately. The output will be printed on the terminal.

- Dependency merge:
  ```
  $ ./AE/scripts/dependency-merge.sh
  ```
- Latency target computation:
  ```
  $ ./AE/scripts/latency-target-computation.sh
  ```
- Priority scheduling:
  ```
  $ ./AE/scripts/priority-scheduling.sh
  ```

B.6.4 Dynamic Provisioning. Based on the results of online scaling (i.e., the number of allocated containers for each microservices), the dynamic provisioning module generates a schedule that assigns the allocated containers to different nodes to balance the interference across the cluster. Please run the script as follows.

```
$ ./AE/scripts/dynamic-provisioning.sh
```

B.7 Evaluation and Expected Results

Erms is evaluated in different aspects. The data will be save in *./AE/data/*. First, we evaluate resource efficiency and the end-to-end latency of online services under static workload as follows.

- Compute theoretical resource allocation first:
  ```
  $ ./AE/scripts/theoretical-resource.sh
  ```
- Generate static workload:
  ```
  $ ./AE/scripts/static-workload.sh
  ```

Next, we evaluate the dynamic workload as follows.

```
$ ./AE/scripts/dynamic-workload.sh
```

In addition, we independently evaluate all modules of Erms, including latency target computation, priority scheduling, and interference-aware provisioning.

- The optimization of latency target computation:
  ```
  $ ./AE/scripts/opt-latency-target-computation.sh
  ```
- Estimate the benefit of priority scheduling:
  ```
  $ ./AE/scripts/benefit-priority-scheduling.sh
  ```
- Estimate benefit of interference-based scheduling:
  ```
  $ ./AE/scripts/interference-scheduling.sh
  ```

B.8 How to Reuse Erms beyond the Paper

In this part, we introduce some tips about how to reuse Erms.

- The project is separated into different modules. Users could redesign each individual module to incorporate other solutions. For example, users can modify latency target computation to design a new algorithm for resource allocation.
- For readability, we use Yaml to configure Erms parameters. Users can modify Yaml files instead of code to easily run Erms under different configurations.

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] Docker containers. 2022. https://www.docker.com/.
[10] Christina Delimitrou and Christos Kozyrakis. 2013. IBench: Quantifying interference for datacenter applications. In *Proceedings of IISWC*.
[11] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of ASPLOS*.
[12] 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*.
[13] Alibaba Cloud Microservices Engine. 2022. https://www.alibabacloud.com/product/microservices-engine.
[14] Google Kubernetes Engine. 2022. https://cloud.google.com/kubernetes-engine.

[15] Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees. 2022. https://doi.org/10.5281/zenodo.7220659.

[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] 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.*

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

[24] 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.*

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

[26] 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.*

[27] 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.*

[28] 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.*

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

[30] 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.*

[31] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. 2021. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *Proceedings of SOSP.*

[32] 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.*

[33] 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.*

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

[35] 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.*

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

[37] 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.*

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

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

[40] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed resource management across process boundaries. In *Proceedings of ACM SoCC.*

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

[42] 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.*

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

[44] 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.*

[45] 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.*

[46] 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.*

[47] Yang Zhou, Hassan MG Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E Culler, Henry M Levy, et al. 2022. Carbink:Fault-Tolerant Far Memory. In *Proceedings of OSDI.*