Embracing Imbalance: Dynamic Load Shifting among Microservice Containers in Shared Clusters

Shutian Luo*
Yale University
New Haven, USA
shutian.luo@yale.edu

Jianxiong Liao*
Sun Yat-sen University
University of Macau
China
liaojx9@mail2.sysu.edu.cn

Chenyu Lin
University of Macau
Macau SAR, China
mc14889@connect.um.edu.mo

Huanle Xu†
University of Macau
Macau SAR, China
huanlexu@um.edu.mo

Zhi Zhou†
Sun Yat-sen University
Guangzhou, China
zhouzhi9@mail.sysu.edu.cn

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

Abstract

In a unified resource scheduling architecture, containers within the same microservice often encounter temporal and spatial performance imbalance when deployed in large-scale shared clusters. As a result, the commonly employed load-balancing approach often leads to substantial resource wastage as applications are frequently over-provisioned to meet service level agreements (SLAs).

In this paper, we utilize an alternative approach by leveraging load imbalance. The central concept involves the dynamic load shifting across microservice containers with a focus on imbalance awareness. However, achieving seamless integration between load shifting and resource scaling, while accommodating the demands of partial connection between upstream and downstream containers, remains a challenge. To address this challenge, we introduce Imbres—a new microservice system that optimizes load shifting, connection management, and resource scaling in tandem. One significant advantage of Imbres lies in its rapid responsiveness, relying solely on online gradients of latency, eliminating the need for offline profiling. Evaluation using real microservice benchmarks reveals that Imbres reduces resource allocation by up to 62% and decreases SLA violation probability by up to 82%, compared to state-of-the-art systems.

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

*Co-first author. Both authors contributed equally to this paper.
†Corresponding authors.

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/2025/03
https://doi.org/10.1145/3676641.3716255

Keywords: Load Shifting, Microservice Scaling, SLA Guarantees

ACM Reference Format:
Shutian Luo, Jianxiong Liao, Chenyu Lin, Huanle Xu, Zhi Zhou, and Chengzhong Xu. 2025. Embracing Imbalance: Dynamic Load Shifting among Microservice Containers in Shared Clusters. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716255

1 Introduction

Microservices, due to their lightweight and granular nature, are quickly replacing monolithic applications and becoming the preferred choice for production data centers. Unlike monolithic applications, microservice architectures consist of multiple fine-grained components for each application, offering a broad range of advantages, including streamlined deployment, faster development, and efficient resource management [5, 18, 48, 53, 59]. Major internet companies are widely implementing microservices. For instance, Meta and Alibaba have deployed over ten thousand microservices in their data centers, collectively handling millions of microservice invocations on a daily basis [21, 32].

Meanwhile, in modern data centers, there has been a shift towards a unified scheduling paradigm using a single scheduling framework [30]. One distinct aspect of this paradigm is its ability to enhance the overall resource utilization by scheduling both latency-sensitive applications, such as microservices, and best-effort applications, such as batch processing jobs, on the same physical hosts [9]. However, this new paradigm also leads to an uneven distribution of resource usage across physical hosts due to the resource-intensive nature of batch processing tasks, resulting in highly dynamic and unpredictable resource interference that affects latency-sensitive applications [31, 32]. Our analysis of Alibaba production traces has revealed a significant resource utilization difference of up to 3.8× within the same host over time and

an even higher difference of up to 5.2× across physical hosts during the same period (Fig. 1 in § 2.1). Consequently, performance imbalance among containers occurs [12], highlighting the need for a tailored solution to effectively balance performance for each microservice in interfered environments.

Load shifting, which redistributes invocations among containers, emerges as a promising approach to balance microservice performance. This load imbalance approach holds great potential for optimizing overall resource usage, as exemplified in the motivation for load shifting (Fig. 4 in § 2.3). While load imbalance has been extensively studied in performance optimization for monolithic applications, employing adaptive, pull-based, push-based, and feedback-based load balancing strategies [7, 13, 14, 27, 40, 45, 51, 54, 59], the complex microservice architecture introduces new challenges at multiple levels. At the intra-microservice level, it is common for an individual microservice to be allocated hundreds to thousands of containers, each experiencing diverse levels of resource interference within a unified scheduling framework. Consequently, this leads to an expansive search space for identifying the optimal load shifting among containers to achieve balanced performance. At the inter-microservice level, containers establish only partial connections through remote procedure calls (RPC) between upstream and downstream microservices [11, 26]. These partial connections can exacerbate performance imbalance, particularly for the connections between highly interfered containers. As a result, these connections need to be jointly managed alongside load shifting, introducing additional complexity to the system. At the application level, scaling containers is crucial for meeting stringent end-to-end (E2E) latency requirements in dynamic environments. However, scaling can alter the connections between microservice containers, necessitating extra load shifting to maintain performance.

In this paper, we introduce Imbres, an Imbalance-aware resource scaling framework that mitigates the uneven performance interference in microservice applications by leveraging proactive load imbalance. To effectively address the aforementioned challenges, Imbres employs a three-tiered system. For each microservice, Imbres incorporates a lightweight iterative load shifting process, which adopts a specialized variant of the gradient descent method to rapidly adjust loads between containers. Guided by real-time feedback on microservice latency, this process ensures rapid convergence while maintaining balanced performance across all containers within each microservice. For each pair of dependent microservices, Imbres deploys a scalable connection establishment mechanism, effectively managing connections and load distributions between related containers. This mechanism alleviates contentions by categorizing upstream microservices into groups based on a performance hash table, and then applying an efficient algorithm to establish connections between upstream and downstream containers on a group-by-group basis. At the application level, Imbres adopts a hierarchical approach to scale containers in response to intensified resource interference or significant changes in invocation arrival rates. This approach begins with a numerical optimization based on Newton's method, which calculates the global amount of resources to be scaled for the entire application. Subsequently, an iterative method utilizing the performance hash table is employed to determine the optimal amount of resources to be scaled for each microservice. More importantly, Imbres also seamlessly integrates dynamic load balancing, connection management, and resource scaling, fostering optimal synergy among these pivotal operations.

Imbres is deployed on both a local cluster and the AWS cloud, and we conduct comprehensive experiments using the DeathStarbench [18] and Alibaba production traces [1]. The outcomes reveal Imbres' effectiveness, showcasing up to a 62% reduction in resource allocation and a noteworthy 82% decrease in SLA violation probability compared to state-of-the-art systems. In summary, our contributions are threefold: we introduce a **novel load control mechanism**, where Imbres dynamically shifts loads and adjusts connections among containers to mitigate resource interference imbalance in microservice applications; we achieve **scalable joint optimization** by concurrently optimizing load control, connection management and resource scaling using a tiered approach, ensuring both SLA compliance and minimized resource allocation; and we enable **low system overhead** by making real-time decisions based on effective performance gradient measurements, distinguishing Imbres from existing microservice systems that rely extensively on offline performance models.

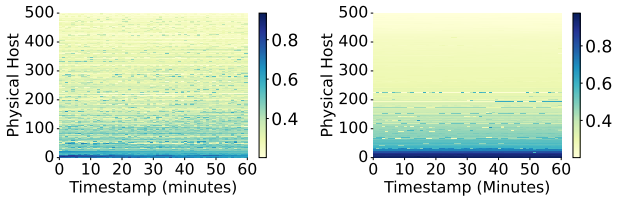
2 Background and Motivation

2.1 Microservice Background

A microservice application typically comprises multiple fine-grained microservices with intricate dependencies [34, 41]. In this section, we explore the characteristics of microservices, including communication between microservices and resource interference external to microservices.

2.1.1 Partially Connected Containers. To achieve high performance, microservices typically communicate with each other via connection-oriented RPC protocols [11, 26]. In such protocols, a persistent connection is established between upstream and downstream microservices. However, maintaining these connections requires significant resources, including memory for connection states and data buffers. This overhead becomes particularly substantial in environments where a microservice might interact with hundreds to thousands of containers across multiple upstream and downstream services within a production cluster.

To reduce the overhead of full connection, each upstream container maintains connections with only a subset of downstream containers. Partial connection is a common practice in production clusters, adopted by companies like Google [52]



(a) CPU resource utilization **(b)** Memory resource utilization

Figure 1. Dynamic variations in resource utilization observed across physical hosts within Alibaba clusters.

and Alibaba [29] as a best practice. The analysis of Alibaba's system traces reveals that, on average, each upstream container connects to only ten downstream containers, chosen uniformly at random [29]. Our experiments also demonstrate that when connected to 200 downstream containers, fully connected setups increase resource usage by 18% and E2E latency by 61% compared to partial connection management.

2.1.2 Dynamic Resource Interference. Resource scaling decisions are typically transmitted by the microservice framework to the unified cluster scheduler [30, 49], which then determines the placement of the requested containers on designated physical hosts. In pursuit of enhanced resource utilization, the unified scheduler frequently consolidates online applications and best-effort jobs within the same physical host to share resources. However, this practice can lead to potential resource interference for microservices.

To scrutinize resource interference, we conduct an analysis of the Alibaba trace [32], utilizing approximately 500 physical hosts over a one-hour period, as depicted in Fig. 1. The results reveal a considerable variance in resource utilization across different hosts, ranging from 10% to 80%. Notably, within the same host, resource utilization is dynamic across various time slots. Specifically, in terms of CPU utilization, even physical hosts with low average utilization as indicated in the top part of Fig. 1 can experience sudden spikes, surging from 10% to 80% within a single minute.

This highly dynamic resource interference often translates into performance imbalance among containers affiliated with the same microservice. To quantify this, we compute latency variance across containers per microservice and assess interference imbalance using resource utilization variance across all hosts. And we further calculate the Spearman correlation between latency variance and interference imbalance. Our investigation reveals varying degrees of performance sensitivity to imbalanced resource interference among distinct microservices, as depicted in Fig. 2. Notably, approximately 50% of the microservices within Alibaba clusters exhibit a strong positive correlation between latency variance and imbalance in resource interference, signifying their susceptibility to performance impacts caused by dynamic interference.

2.2 Variability in Tail E2E latency

Owing to the performance imbalance among containers from individual microservices within a shared environment, the

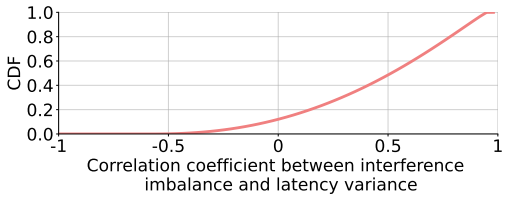
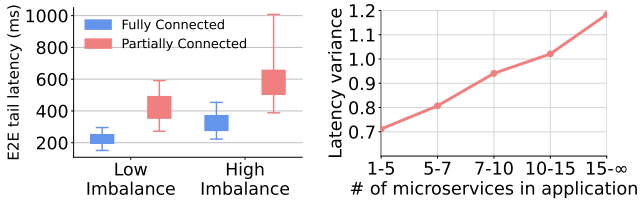


Figure 2. Cumulative distribution function (CDF) depicting the correlation coefficient between resource interference imbalance and variance of microservices' latency.



(a) **(b)**

Figure 3. (a) E2E Latency distribution under different connection schemes and imbalance settings. (b) Variability in E2E latency grows with application complexity.

E2E latency of the entire microservice application can undergo substantial fluctuations. To investigate this phenomenon, we deploy the Social Network application from DeathStarbench [18] on a local cluster with three hosts—two experiencing low resource interference and one subjected to high interference. We apply the default load balancing strategy to evenly distribute load among containers. We then use iBench [16] to generate best-effort jobs with varying resource requirements, ranging from 10% to 60% of CPU resources, memory bandwidth and network bandwidth. We categorize the resource utilization difference between physical hosts as either high imbalance (exceeding 30%) or low imbalance (below 30%). Finally, we calculate the 95th percentile E2E latency for all applications.

As illustrated in Fig. 3(a), the tail E2E latency under a fully connected strategy is, on average, 50% lower compared to the partially connected approach, where each upstream container randomly establishes connections with 10 downstream containers. With partial connection, an upstream container may send numerous invocations to other downstream containers that are also experiencing high interference, potentially leading to prolonged delays. Fig. 3(a) also reveals that the performance disparity increases with the level of interference, underscoring the critical need for more strategic connection management between containers.

Fig. 3(b) reveals that the variability in tail E2E latency within microservice applications increases as the application complexity grows, measured by the number of involved microservices. This pattern is attributed to the nature of a request traversing through all microservices constituting an application, where the cumulative impact of uneven performance caused by imbalanced interference and poor connection becomes evident. Notably, microservice traces have shown that a complex microservice call graph can comprise

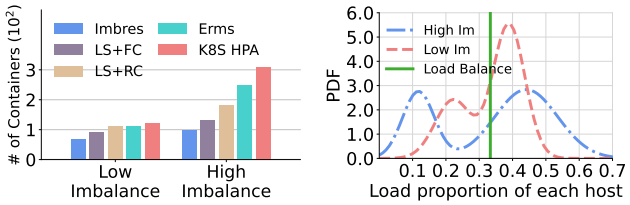


Figure 4. (a) Resource allocation across all schemes, where LS+RC (LS+FC) denotes incorporating load shifting and random partial (full) connection strategy to K8S. (b) Load distribution among different hosts before/after load shifting.

up to 1500 individual microservices [1]. In such cases, the variance of tail E2E latency becomes high, posing significant challenges in effective resource scaling that can ensure SLAs. Consequently, addressing performance imbalance among microservice containers emerges as a critical imperative.

2.3 Load Imbalance: Opportunities and Challenges

Existing systems have developed various resource scaling schemes to address imbalanced interference [22, 33, 39, 41, 44, 57, 58]. These systems rely on load balancing, which assigns an equal load across all containers within each microservice.

Opportunities: We propose that load imbalance could offer a more resource-efficient approach. The intuition behind this is that, current scaling methods provision resources uniformly such that the pressure on the straggling containers (with high-interference) is alleviated, wasting capacity on low-interference containers. Load imbalance, however, reduces tail latency by shifting load among containers to achieve balanced performance. To validate this idea, we conducted an additional experiment, similar to the one described in § 2.2. For resource scaling, we utilized Kubernetes' Horizontal Pod Autoscaling (HPA) [8] and the state-of-the-art system Erms [33] as baselines, ensuring that the 95th percentile E2E latency remains within SLA. Specifically, HPA adjusts the resource allocation of microservices based on predefined resource utilization thresholds, while Erms allocates resources based on the globally optimized latency targets of individual microservices. As depicted in Fig. 4(a), the load-shifting approach, when combined with HPA and random partial connection between upstream and downstream containers (LS+RC), achieves an average reduction of 31% and 20% in resource allocation compared to HPA and Erms with default load balancing. This advantage was even more pronounced under high resource interference, where resource allocation decreased by a factor of 29% to 37.5%.

To better understand how resource savings are achieved, we investigate the load proportion allocated to different hosts after load shifting, for each microservice. Specifically, we fit the load proportion data of all microservices using a multimodal Gaussian distribution, as illustrated in Fig. 4(b). When implementing the load shifting strategy, the workload distribution displays a dual-peak pattern, indicating an uneven assignment of workloads to containers experiencing varying levels of resource interference. In scenarios with high imbalance, the peaks of the load distribution become more dispersed to accommodate extremely imbalanced resource interference, shifting significantly away from the central value of 0.33 (the load proportion under the load balancing scheme). The uneven load distribution facilitates balanced performance among containers and significantly reduces the tail latency of each individual microservice, thereby providing opportunities to enhance resource efficiency.

When the random partial connection is replaced with a full connection (LS+FC), an additional 23% reduction in resource allocation can be achieved. Nevertheless, HPA cannot fully leverage the benefits of load imbalance. The integration of fine-grained resource scaling presents another opportunity for optimizing resource efficiency. Illustrated in Fig. 4(a), the amalgamation of load shifting, connection management, and effective scaling, as embodied in Imbres, leads to a 24% reduction in resource allocation compared to load shifting alone, and a 54% reduction compared to using Erms.

Challenges: Effectively leveraging load imbalance while accommodating the demands of partial connection and dynamic resource scaling for complex microservice applications within large-scale production clusters poses a significant challenge, owing to the intricate coupling between these various factors (as emphasized in § 1). Concurrently, profiling the performance of microservices in advance for optimized resource scaling — a practice akin to existing systems [33, 41, 57] — also presents a new challenge under the unified scheduling paradigm. This profiling process can incur substantial system overhead, given the multitude of configurations to account for, including the number of containers, resource interference across different physical hosts, and dynamic load, without considering the complexity introduced by the frequently evolutionary nature of microservices [28].

3 The Imbres Framework

In this section, we illustrate the design ideas behind Imbres and provide an overview of the system architecture.

3.1 Key Design Ideas

I_1:Leveraging a divide-and-conquer approach to handle the intricate coupling between various factors. Imbres employs a three-tiered design to decouple the intricate optimization problem, which involves a dedicated load control optimization, an intelligent connection management mechanism, and a global resource scaling process. These three components work together to enable rapid adaptation to fluctuations in workload and changes in interference. Imbres also incorporates effective coordination between the optimizations at different tiers.

I_2: Applying efficient online gradient-based algorithms to solve complex optimization problems. Microservices are rapidly evolving within production clusters, leading to

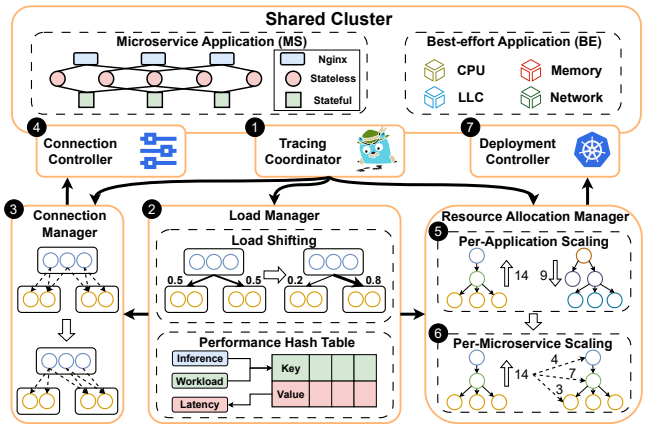


Figure 5. The system architecture of Imbres.

a constantly changing performance profile. To alleviate the heavy profiling overhead, Imbres employs an iterative online learning approach to govern both load control and resource scaling, utilizing performance gradients derived from real-time latency feedback. Furthermore, to expedite this iterative process, Imbres constructs a performance hash table that can effectively memorize all historical control outcomes.

3.2 System Overview

The system architecture of Imbres, illustrated in Fig. 5, encompasses the following key components:

The *Tracing Coordinator* ❶ gathers data from two prominent tracing systems, Prometheus [3] and Jaeger [2]. It specifically extracts information on the latency of individual containers within each microservice and the overall E2E latency of each application from historical traces.

The *Load Manager* ❷ primarily executes the load shifting. It periodically tunes the load distribution across containers within each microservice based on the container latency obtained from the *Tracing Coordinator*. Furthermore, the *Load Manager* maintains a performance hash table for each microservice, serving as a pivotal input for the *Connection Manager* and the *Resource Allocation Manager*.

The *Connection Manager* ❸ operates in tandem with the *Load Manager* and takes the responsibility of adjusting the RPC connections between upstream and downstream containers. It determines the partial connection relationships among containers and the load proportion allocated to each connection, based on the load distribution provided by the *Load Manager* and the performance of upstream containers attained from the performance hash table. The resulting connection adjustments are then applied to related containers through the *Connection Controller* ❹, executing the actual connection establishment and load control.

The *Resource Allocation Manager* mainly executes the resource scaling process, comprising two key modules: *Per-Application Scaling* ❺ and *Per-Microservice Scaling* ❻. It adjusts the number of containers for all microservices in response to changes in load and resource interference when the control of the *Load Manager* alone cannot satisfy SLA requirements. Specifically, the *Per-Application Scaling* module utilizes the E2E latency as feedback to compute the number of containers to be scaled for the application. Subsequently, the *Per-Microservice Scaling* module further assigns these scaled containers to specific microservices. The final scaling results are deployed through the *Deployment Controller* ❼.

4 Design Details of Imbres

4.1 Load Control

The *Load Manager* shifts loads across a substantial number of containers based on the load and interference conditions, guided by the feedback of microservice-level latency.

4.1.1 Gradient-based Load Shifting. The core objective of load shifting is to achieve uniform latency distribution across all containers within each microservice, leveraging the connections established by the *Connection Manager*. Because containers residing on the same physical host encounter identical resource interference, an equal amount of load is allocated to each container within the same hosts to uphold consistent performance. As a result, adjustments in load proportions are solely made on distinct physical hosts. This simplification effectively reduces the complexity of the load shifting, as the number of physical hosts is significantly smaller than that of microservice containers.

For a specific microservice, l_j denotes the average latency of its containers within a host $j \in \{1, 2, \cdots, N\}$ where N is the total number of hosts hosting this microservice. Additionally, l_{mean} represents the average latency of all its containers across N hosts. The load shifting decision D_j for this specific microservice, represents the proportion of load distributed to containers on host j. Then, to achieve uniform latency distribution across N hosts, the load shifting problem can be cast as follows:

$$\min_{\{D_1, D_2, \cdots, D_N\}} \sum_{j=1}^{N} (l_j - l_{mean})^2, \quad \text{s.t.,} \sum_{j=1}^{N} D_j = 1. \tag{1}$$

In production clusters typically with thousands of physical hosts, considering the continuously evolving microservices and dynamic loads, the latency of containers on each host is a function of runtime factors, such as load and the associated resource interference. Therefore, it is highly challenging to derive a closed-form formula of latency. To address this challenge, heuristic solutions, such as genetic algorithms [20], iteratively solve the optimization problem (1) based on feedback generated from trials without relying on pre-defined latency formulas. However, these methods often require a large number of iterations.

In contrast, Imbres employs a distinctive approach by adopting a derivative-free optimization method. In particular, online gradient ascent [46] is utilized in this method, extracting gradient information directly from approximations rather than depending on a predefined formula. The process entails calculating the disparity between the actual

Algorithm 1: Load Shifting Algorithm

Procedure `LoadShifting`(*learning rate* $\eta > 0$):
  Initialize load proportion D_j with load balance;
  for $t = 0, 1, 2, \ldots$ **do**
    /* l_j^k represents the latency of container k located on host j */
    $l_j = \frac{\sum_k^{S_j} l_j^k}{S_j}$ for host $j \in \{1, 2, \cdots, N\}$;
    $l_{mean} = \frac{\sum_j^N l_j}{N}$;
    $F = \{F_j \mid 0 \le j \le N;\ F_j = -1 \text{ if } l_j > l_{mean} \text{ else } 1\}$;
    $\hat{l}_j = \frac{|l_{mean} - l_j|}{l_{mean}}$ for host $j \in \{1, 2, \cdots, N\}$;
    $D_j = D_j * (1 + \eta \cdot F_j \cdot \hat{l}_j)$ for host $j \in \{1, 2, \cdots, N\}$;
    $D_j = \frac{D_j}{\sum_j^N D_j}$ for host $j \in \{1, 2, \cdots, N\}$;

latency on each host and the average latency across all hosts, expressed as $\left|l_j - l_{mean}\right|$, and employing this difference (normalized by the average latency) as an approximated gradient to steer the optimization process. This straightforward yet effective approach demonstrates efficiency in discerning the required gradient for facilitating load-shifting policies, harmonizing seamlessly with the dynamic nature of the cluster.

The entire control process is encapsulated in Algorithm 1, which illustrates an iterative approach to adjusting load distribution. For each specific microservice, at the initial timestamp $t = 0$, container loads are initially balanced by initializing the load proportion D_j to be $S_j / \sum_j^N S_j$ for each host j, where S_j is the number of containers residing on host j. Throughout each five-second tuning interval, the *Load Manager* compares l_j and l_{mean} to determine the update direction F_j and compute the latency disparity $\hat{l}_j$. Subsequently, the *Load Manager* computes a gradient-ascent update in the direction of $F_j \cdot \hat{l}_j$, and scales it by a learning rate η, which is set at a constant 0.2. In cases where a significant difference exists between individual container latency and the average latency, a larger step size is assigned to update the load proportion more assertively. Imbres normalizes the load distribution D_j to ensure the sum of load proportions equals 1. The algorithm boasts low complexity, rendering it easy and efficient for implementation in a production cluster.

4.1.2 Construction of Performance Hash Table.

The performance hash table is generated by the *Load Manager* during the load shifting process, storing mappings from a pair of keys comprising resource interference and load per container to latency for quick lookup.

Specifically, Imbres considers three types of resource interference [1] within the first key, defined as follows:

$$I_i = (CPU_i, MemB_i, Net_i), \tag{2}$$

where CPU_i, $MemB_i$, and Net_i are the average utilization of CPU, memory bandwidth, and network bandwidth across all physical hosts running containers of a given microservice i. The second key w_i quantifies the average workload per container handled by microservice i. Moreover, the value L_i associated with each pair of keys represents the tail latency of all containers within microservice i. Then, the mapping operation between the pair of keys (I_i, w_i) and value L_i can be expressed as follows:

$$\phi_i(I_i, w_i) \longrightarrow L_i. \tag{3}$$

Each type of interference is categorized into four equally sized groups, ranging from 0% to 80%. The second key w_i is also divided into four equal categories from zero to the maximum load. Here, the maximum load is the value at which a container's CPU utilization hits 50% without interference. Estimating the maximum load in this manner is justified because CPU utilization for the majority of microservices in production clusters is typically below 50% [1, 50].

The keys and values are collected during the execution of each microservice container and updated in runtime. When querying the hash table, if the first key is not found, the value corresponding to the closest key is returned. In contrast, if the second key w_i is absent, Imbres leverages linear interpolation to construct the value. In this case, the two closest keys, denoted by w_m and w_n (where $w_n > w_m$), are selected, and the result of this querying can be computed as follows:

$$\phi_i(I_i, w_i) = \frac{\left(\phi_i(I_i, w_n) - \phi_i(I_i, w_m)\right) \cdot (w_i - w_m)}{w_n - w_m}. \tag{4}$$

To strike a balance between storage efficiency and the accuracy of latency estimation, Imbres employs four categories by default for the performance hash table. We provide a detailed analysis of this choice in § 6.6.

4.2 Connection Management

The *Connection Manager* establishes partial connections between upstream and downstream containers for each pair of microservices, ensuring that the load received by each downstream container aligns with the outcomes of load shifting. Simultaneously, it enforces the total number of connections within a specified threshold to reduce resource overhead.

4.2.1 A Linear Integer Programming Formulation.

To reduce the number of connections between upstream and downstream containers, a straightforward approach involves formulating a linear integer programming problem. In this formulation, the constraint is to regulate the load on each connection, ensuring that the resulting load assigned to each downstream container aligns with the load distribution generated by the *Load Manager*. However, directly employing this approach may result in a significant degradation in E2E latency. This is primarily due to the varying degrees of resource interference experienced by upstream containers, which affects their ability to process requests consistently. As a result, these containers dispatch invocations to downstream containers at rates that do not align with their assigned loads until the load shifting process of the upstream microservice reaches convergence. Consequently, solutions

that neglect the imbalanced performance of upstream containers may introduce mismatches between invocation arrival rates and load distribution in downstream containers, potentially leading to violations of SLAs. To address this issue, a naive remedy would be to delay the load shifting process of the downstream microservice until the upstream microservice achieves convergence. However, this approach can cause delays in subsequent microservices' load shifting processes, particularly in scenarios with long chains of interconnected microservices.

4.2.2 Grouping-based Connection. To mitigate the impact of imbalanced performance among upstream containers during the connection management process, we propose a grouping-based approach. This approach first leverages the performance hash table to categorize upstream containers based on their processing capabilities. Containers within the same group exhibit similar performance, allowing the *Connection Manager* to treat them as a collective unit for connection management. Subsequently, connections are established leveraging an efficient algorithm between each entire group and all downstream microservice containers.

The *Connection Manager* utilizes the current load of the upstream microservice, load proportion in the current iteration, and resource interference of each upstream container to query the performance hash table for the corresponding latency information. Subsequently, a K-means algorithm is employed to categorize all upstream containers into different groups based on their respective latency values. The number of groups is determined by dividing the specified connection threshold by the number of downstream containers.

Using the group information, the *Connection Manager* establishes connections on a group-by-group basis. Since the invocation delivery rates across upstream containers within the same group exhibit similar behavior, the *Connection Manager* can manage the connections of all groups and all pairs of microservices in parallel. For each group, a linear integer programming problem is formulated to minimize the number of connections, as stated in § 4.2.1. To solve this problem while preventing less-performing upstream containers from being connected to downstream containers suffering from high resource interference, the *Connection Manager* employs an efficient heuristic. Specifically, it sorts the upstream containers within each group and all the downstream containers according to the load proportion assigned by the corresponding load shifting processes in the current iteration. Subsequently, the *Connection Manager* follows the principle of connecting an upstream container with the highest load proportion (the values are normalized within each group such that they sum up to one) to a downstream container with the least load proportion. Once the remaining load proportion of the selected upstream container cannot match the load proportion of the selected downstream container, the *Connection Manager* selects another upstream container and connects it to this downstream container. These connections are established one by one until all downstream containers have their load proportions fully matched. Moreover, each upstream container sends invocations to its connected downstream containers in accordance with the matched load proportions.

4.3 Dynamic Resource Scaling

For efficient resource scaling, existing works primarily construct analytical models or employ machine learning techniques to establish accurate latency-resource mappings for microservice applications [22, 33, 58]. However, achieving this mapping becomes challenging in the presence of highly dynamic and imbalanced resource interference. Moreover, the complexity of dynamic load shifting in our scenarios makes it difficult to enumerate all possible load configurations for such a latency-resource mapping. Instead, Imbres adopts an alternative approach by scaling resources based on E2E latency feedback using an iterative methodology.

At each time interval, the *Per-Application Scaling* module employs an online learning method to calculate the number of containers to be scaled, denoted as dR, for the entire microservice application. This value, dR, is subsequently utilized as an input for the *Per-Microservice Scaling* process. If dR is non-zero, it indicates that the application requires scaling up or down of resources. In such cases, the *Per-Microservice Scaling* module is responsible for assigning these scaled containers to critical microservices. The identification of these critical microservices relies on performance hash tables.

4.3.1 Per-application Resource Scaling. The primary objective of *Per Application Scaling* is to determine an appropriate number of containers, denoted as R, to allocate for an application while meeting the SLA requirements, based on the current load W of the entire application. Although different microservices within an application may have diverse requirements for the resource configuration of containers, minimizing the number of allocated containers generally results in more efficient resource allocation.

The fundamental insight guiding Imbres' resource scaling is that the tail E2E latency L generally increases monotonically with a rise in the load-resource ratio $P = \frac{W}{R}$. This trend is observed because an increased load on a container tends to lead to a longer average queuing time due to limited computing resources, aligning with queuing theory [24]. Consequently, the *Per Application Scaling* module concentrates on maximizing the ratio P, while adhering to the SLA constraints, as illustrated in Algorithm 2.

In a manner akin to the load-shifting policy, the *Per Application Scaling* module adjusts the load-resource ratio P_t within the current time interval t, striving to minimize the difference between the E2E latency L_t and SLA. The central concept involves employing an online gradient-ascent-based approach, using the E2E latency of the application as the sole feedback. To expedite convergence, the module incorporates Newton's method to compute the gradient when the

Algorithm 2: Per Application Scaling Mechanism

Function `Per_APP_Scaling`(R, L, SLA, W):
  /* Compute P_t, the load-resource ratio at time t */
  $P_t = \frac{W_t}{R_t}$;
  /* L_t represents the E2E latency at time t */
  if $(L_t - L_{t-1}) * (P_t - P_{t-1}) > 0$ & $|L_t - L_{t-1}|/L_t >= \beta$ **then**
    $dP = \eta_1 * \frac{(P_t - P_{t-1})(SLA - L_t)}{L_t - L_{t-1}}$;
  else
    $dP = \eta_2 * \frac{(SLA - L_t) * P_t}{L_t}$;
  $P_t = P_t + dP$;
  $R^* = \frac{W_t}{P_t}$;
  return $(dR = R^* - R)$;

Algorithm 3: Per Microservice Scaling Mechanism

Function `Per_MS_Scaling`(dR, C, ϕ, I, D, N):
  $f = 1$ if $dR > 0$ else -1;
  while $dR \neq 0$ **do**
    /* D_j^i is the load of microservice i distributed to host j; N_i is the number of hosts that are hosting microservice i */
    $\sigma_i = \sqrt{\frac{\sum_{j=1}^{N_i} \left(D_j^i - \sum_{j=1}^{N_i} D_j^i / N_i\right)^2}{N_i}}$;
    $\alpha_i = 1 + \sigma_i \cdot N_i \Big/ \sum_{j=1}^{N_i} D_j^i$;
    $G_i = \alpha_i \cdot \left(\phi_i(I_i, \frac{\rho_i}{C_i}) - \phi_i(I_i, \frac{\rho_i}{C_i + f})\right)$;
    $k = \arg\max_i G_i$;
    $C_k = C_k + f$;
    $dR = dR - f$;
  return C;

difference between two successive E2E latency, $L_t - L_{t-1}$, exceeds a threshold of $\beta \times L_t$ (β set at 0.1 by default). During the Newton's update, a large learning rate η_1 is applied, for instance, set at 0.5.

However, there are cases where the latency does not strictly increase with the load-resource ratio P_t due to measurement noise, or when the change in E2E latency is marginal. Directly applying Newton's method [23] can result in a gradient explosion that impedes the convergence. To address this, the *Per Application Scaling* module employs conventional first-order gradient descent [46] with a smaller learning rate $\eta_2 = 0.2$ to perform the update. Upon the update of the ratio P_t, the number of scaled containers can be easily computed.

4.3.2 Per-microservice Resource Scaling.

Per Microservice Scaling module is integrated to allocate the number of dR containers to different microservices within the application. In the absence of an exact performance model, this module accomplishes the allocation through an iterative process that strategically selects a critical microservice in each iteration. The assessment of criticalness is mainly based on two key aspects: the sensitivity of an individual microservice's latency to resource allocation and the extent of load imbalance among all containers within each microservice after the operation of the load shifting, as detailed in Algorithm 3. The extent of load imbalance is calculated using the variance of load distribution. A higher variance suggests that the microservice is more sensitive to resource interference and experiencing higher resource imbalance. Allocating more containers to such a microservice can help mitigate the negative impact of interference.

The evaluation of microservice i's sensitivity relies on the performance hash table ϕ_i constructed by the *Load Manager*. This table ϕ_i maps the pair $(I_i, \frac{\rho_i}{C_i})$ to latency for microservice i, where ρ_i represents the load of the whole microservice, C_i is the number of its containers, and I_i denotes the average resource interference on the physical hosts hosting its containers. The details of creating this table are described in § 4.1.2. Using this table, the reduction in latency for microservice i when adding one extra container can be expressed as follows:

$$g_i = \phi_i\left(I_i, \frac{\rho_i}{C_i}\right) - \phi_i\left(I_i, \frac{\rho_i}{C_i + 1}\right). \tag{5}$$

On the contrary, the change in latency for microservice i when reclaiming one container can be expressed as:

$$g_i = \phi_i\left(I_i, \frac{\rho_i}{C_i}\right) - \phi_i\left(I_i, \frac{\rho_i}{C_i - 1}\right). \tag{6}$$

The *Per-Microservice Scaling* module computes σ_i, the variance of load distribution across different physical hosts, and normalizes it by the average load to obtain the load imbalance index α_i. The criticality score of a microservice is then determined by $G_i = \alpha_i \cdot g_i$. The module selects the microservice that generates the highest G_i for resource scaling. It is worth noting that, in each iteration, only one container is added or reclaimed from the most critical microservice. To reduce the computational complexity, the criticality of microservices G can be stored in a priority queue. Only the criticality of the chosen microservice will be updated and pushed into the queue, resulting in a computational complexity of $O(dR \cdot \log m)$, where m is the number of microservices within a microservice application.

5 Imbres Implementation

The Imbres system is implemented on top of Kubernetes [25], and its development facilitates the Kubernetes Python client library, comprising 3,500 lines of Python code. The *Tracing Coordinator* periodically queries Jaeger [2] to extract real-time latency data, with trace collection resource usage typically remaining below one CPU core and 2 GB memory for each application. Additionally, it queries Prometheus [3] to obtain interference data at each host.

We store the performance hash table built by the *Load Manager* in Redis [42] and design an interface for Imbres operators to specify the number of key groups categorized by the performance hash table. We have developed the *Connection Controller* as a component responsible for implementing the connection strategy generated by the *Connection Manager* within the network layer. The *Connection Controller*

assigns load proportion to different connections between upstream and downstream containers. The load assignment is accomplished using Linux `iptables`, achieved by modifying forwarding rules and assigning load forwarding probabilities to each connection based on the virtual IP addresses of containers. The `iptables` update latency is typically less than 10 milliseconds. The *Resource Allocation Manager* periodically synchronizes the performance hash table with the *Load Manager* to enhance the accuracy of latency estimation. To deploy containers, we utilize the default deployment controller in Kubernetes to scale the number of containers.

6 System Evaluation

6.1 Experimental Setup

Benchmarks. We evaluated Imbres using the DeathStarbench [18] as well as the Train Ticket [60] benchmark. DeathStarbench includes three applications: Social Network, Media Service, and Hotel Reservation.

System setup. We evaluated Imbres with DeathStarbench on a local cluster comprising 4 two-socket physical hosts. For the more complex and resource-intensive Train Ticket application, we conducted the evaluation on an expanded cluster with 16 two-socket physical hosts. Each host in the local cluster is equipped with 100 Intel Xeon CPU cores and 128GB of RAM. We also assessed Imbres on 17 Amazon EC2 c6a.8xlarge instances, with each virtual machine configuration comprising 32 CPU cores and 64 GB of memory [4]. Unless otherwise specified, each microservice container is configured with 0.1 core, 200MB of memory for DeathStarbench, and 0.2 core, 500MB of memory for Train Ticket. The iterations within the load control of Imbres are performed every 5 seconds, with a detailed analysis of parameter tuning provided in § 6.6. Resource scaling is executed every one minute, aligning with the typical resource scaling interval used in production clusters. Moreover, the total number of connections between upstream and downstream containers within each pair of microservices is capped at ten times the total number of downstream containers, in alignment with the settings commonly used in production clusters [29].

Environment setup. We run experiments with both static and dynamic workloads, setting SLA targets based on the 95th percentile E2E latency. For DeathStarBench applications, we establish SLA targets ranging from 100 ms to 500 ms, defining a *strict SLA* for targets below 300 ms and a *slack SLA* for those exceeding 300 ms by default. We generated static workloads at 1200 requests per second, the maximum throughput our cluster can support under high interference imbalance. For Train Ticket, SLA targets range from 1000 ms to 2000 ms, with a *strict SLA* for targets below 1500 ms and a *slack SLA* for those exceeding 1500 ms. We generated static workloads at 200 requests per second.

To mimic diverse resource interference scenarios, we generated interference in terms of CPU, memory bandwidth, and network bandwidth using iBench [16]. This was achieved by varying the number of anomaly containers on each target host, thereby diversifying the resource utilization of hosts from 10% to 70%. In the entire cluster, the average host utilization hovers around 50%, with anomaly containers accounting for 35% and microservice containers for 15%. This configuration aligns with the typical setup of production clusters at Alibaba (details are described in § 2.1.2).

Baseline schemes. We compared Imbres against state-of-the-art systems, including Erms [33], Rhythm [58], and Firm [41]. Erms profiles the latency of microservices and fits it with piece-wise linear functions. It computes latency targets assigned to different microservices based on function profiles, considering the microservice dependencies. In contrast, Rhythm assigns latency targets to different microservices based on the normalized product of mean latency, variance of latency across different workloads, and the correlation coefficient between microservice latency and E2E latency. Firm applies an SVM model to identify the critical microservice. Then, it employs reinforcement learning to adjust the resources allocated to these critical microservices.

6.2 Resource Efficiency and E2E Latency

Static workload. We first report the results for all baseline systems under different resource interference levels and SLA settings. To mitigate random noise, each setting was evaluated five times in our private cluster, and the average of these evaluations was computed as the final result.

As depicted in Fig. 6, across all test scenarios, Imbres achieves an average reduction in resource allocation of 49%, 61%, and 53% compared to Erms, Rhythm, and Firm, respectively. Rhythm relies on static microservice latency statistics for resource allocation, it struggles to adapt to dynamic latency variations. Firm, on the other hand, concentrates exclusively on critical microservices, the reinforcement learning agent cannot effectively address multiple critical microservices simultaneously. Erms attempts to address resource interference by strategically placing containers but falls short in addressing performance imbalance under partially connected settings among microservice containers. In contrast, Imbres effectively mitigates these limitations, particularly in scenarios where high resource interference imbalance (Im) results in a resource contention difference exceeding 30% between the corresponding hosts. In such situations, Imbres reduces the number of allocated containers by more than 63% compared to all baselines. This notable improvement is attributed to its adaptivity in balancing latency under high resource interference environments. A similar enhancement is observed in strict SLA settings, where more containers are allocated to meet SLA requirements, amplifying the performance imbalance and variance under dynamic resource interference. Furthermore, compared to all baselines, Imbres achieves average resource savings of 53%, 54%, 49%, and 62% on the Social Network, Media Service, Hotel Reservation, and Train Ticket applications, respectively. Notably, Imbres saves

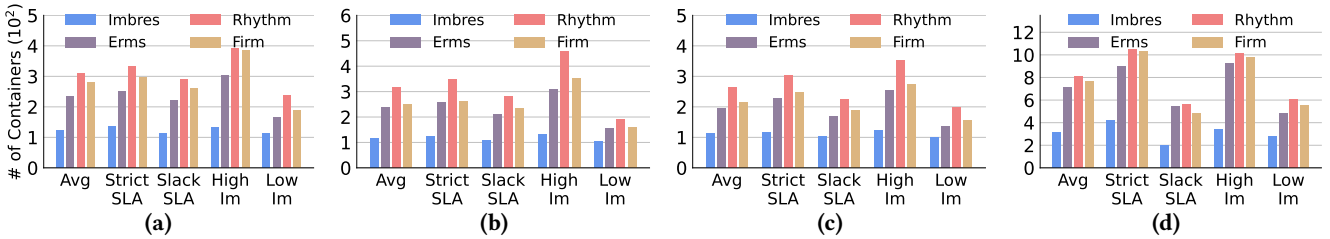


Figure 6. Overall resource allocation. (a) Social Network. (b) Media Service. (c) Hotel Reservation. (d) Train Ticket

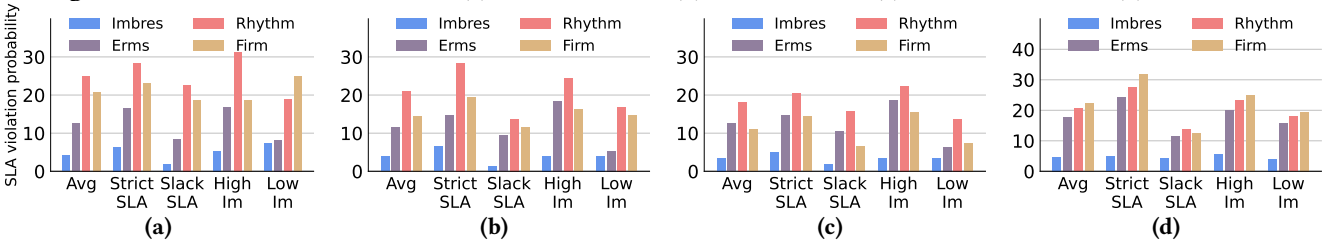


Figure 7. SLA violation probability. (a) Social Network. (b) Media Service. (c) Hotel Reservation. (d) Train Ticket

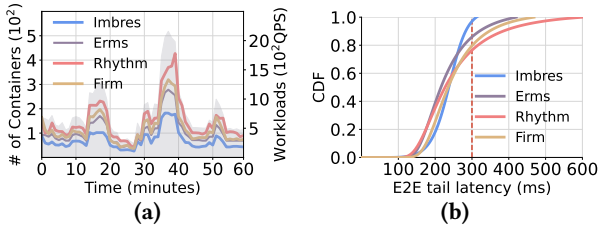


Figure 8. Resource allocation and tail E2E latency of all systems under dynamic workloads. (a) Workload and allocation of containers over time. (b) CDF of E2E latency.

an additional 10% of resources on the Train Ticket application, which features 2× more microservices than the other applications. This result demonstrates Imbres' efficiency in managing complex applications in larger-scale clusters.

Figure 7 depicts the SLA violation probability across different scenarios. On average, Imbres sustains a violation probability of less than 5% for all applications, markedly lower than the rates observed with Erms, Rhythm, and Firm, which stand at 13.6%, 21.2%, and 17.1%, respectively. Additionally, the probability of SLA violations is notably higher for the baseline models in settings with strict SLA requirements and high resource interference imbalance. This underscores Imbres' superior adaptability in such challenging environments.

Dynamic workload. We also deployed Imbres using the SocialNetwork application with dynamic loads derived from Alibaba traces [32], setting the SLA target to 300ms. The shadow area in Fig. 8(a) illustrates the fluctuations of load over time in our private cluster. As depicted in Fig. 8(a), Imbres reduces the overall resource allocation by about 36% compared to Erms and achieves up to 54% savings compared to Rhythm on average. Fig. 8(b) displays the cumulative distribution function (CDF) of the tail latency of requests over time. The result indicates that Imbres consistently meets the SLA requirements, even in the face of rapidly increasing loads. In contrast, other schemes, such as Rhythm, which exceeds the SLA threshold with a probability as high as 20%, often fail to meet SLA requirements due to their inability to address performance imbalance among containers.

6.3 The Rationale Behind Imbres' Design

In this section, we conduct a comprehensive study on the rationale for Imbres' design. Specifically, we aim to explore the factors contributing to the variance in microservice performance and the influence of full connection mechanisms on overall performance, which highlights the critical importance of effective load shifting and connection management.

How does performance vary across containers? To investigate the root causes of performance variations, we deployed the Social Network application across two groups of hosts, each experiencing different levels of resource interference. Specifically, resource interference—quantified by host utilization for CPU, memory bandwidth, and network bandwidth—ranges from 10% to 60%. We defined low interference as resource utilization below 30% and high interference as resource utilization exceeding 30%. We tracked the workload and latency changes of containers within each microservice throughout the whole load-shifting tuning process. At the beginning, the workload is evenly distributed among containers, while resource interference varies across containers and further causes high latency variance. During this process, the workload is continuously shifted from containers experiencing higher interference to those with lower interference, as illustrated in Fig. 9(a). Despite the reduced workload, containers exposed to higher interference still experience significantly increased latency, exceeding 3× even with a 66% reduction in workload compared to containers with lower interference, as shown in iteration 3 of Fig. 9(a) and Fig. 9(b) (highlighted in red boxes).

To thoroughly assess the sources of performance variation among microservice containers, we calculated the Spearman correlation coefficients between the resource interference

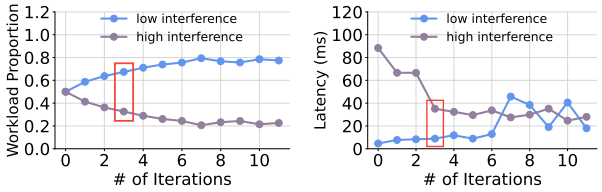


Figure 9. Performance variation during load shifting. (a) Workload proportion of containers. (b) Latency of containers.

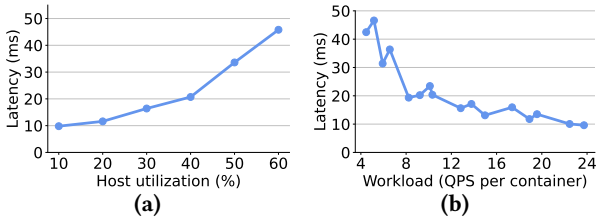


Figure 10. The variation of container latency during the third iteration of the load-shifting process. (a) Container latency under different levels of resource interference. (b) Container latency under varying workloads.

experienced by containers and their corresponding latency. This analysis was performed across all containers in each iteration of the load-shifting process, and the resulting coefficients were then averaged across all iterations. The results reveal that the correlation coefficient between resource interference and latency is 0.87, 0.76, 0.82, and 0.91 for the Social Network, Media Service, Hotel Reservation, and Train Ticket applications, respectively. In contrast, the correlation coefficients between container latency and workloads are negative, specifically -0.62, -0.42, -0.44, and -0.56 for these four applications. These values are significantly lower than the correlations observed between resource interference and latency. The key reason for this negative correlation is that containers experiencing high resource interference tend to exhibit higher latency, prompting a gradual traffic shift to lower-latency containers. Consequently, less-interfered containers are subjected to higher workloads but maintain lower latency, and vice versa, resulting in a negative correlation.

We also examined latency fluctuations across various levels of resource interference and workloads, drawing insights from the results of iteration 3 in the load-shifting process depicted in Fig. 9. The analysis demonstrates that latency increases with higher levels of interference (host utilization), as illustrated in Fig. 10(a). Conversely, container latency consistently decreases as workloads increase for containers experiencing less interference, as shown in Fig. 10(b). These findings highlight resource interference as a critical factor driving performance variation and emphasize the importance of load shifting in mitigating its impact.

Why using partial connection? To quantify the overhead associated with full connection, we deployed the Social Network application with varying numbers of containers in our clusters and generated workloads proportional to the

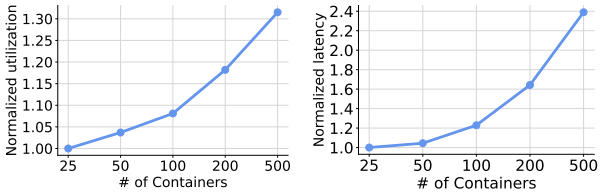


Figure 11. Resource utilization and E2E latency using full connection with varying numbers of downstream containers.

container count. We evaluated the resource utilization and E2E latency across different settings. As the number of downstream containers increases, microservices require additional resources to maintain persistent connections and interact with their downstream microservices. This leads to an increase in overall resource usage and end-to-end latency of the application. Specifically, when the number of containers reaches 500, overall resource usage increases by 31%, as shown in Fig. 11(a), while the E2E latency increases by up to 1.38×, as depicted in Fig. 11(b). These observations underscore the importance of effective connection management using partial connection within large-scale deployments.

6.4 Microscopic Evaluation

In this section, we provide an in-depth analysis of the individual contributions of Imbres's various components by leveraging DeathStarBench applications.

Advantage of Imbres' load shifting. To quantify the improvement brought by dynamic load shifting in complex scenarios, we incorporated the same load shifting policy into all baseline systems while maintaining their original resource scaling strategies and employing a fully connected approach for connection management.

Fig. 12(a) presents the average number of containers allocated under different load control strategies. The results show that all systems improve resource efficiency through dynamic load shifting, demonstrating the importance of addressing interference imbalance. Imbres can reduce resource allocation by 39%, while other schemes can save, at most, 31% of containers. Imbres's superior performance is attributed to its joint optimization between load-control and resource scaling, which not only balances performance among containers for resource savings but also enhances its resource scaling strategy with load variance awareness.

Benefit of Imbres' connection management. We conducted an ablation study of Imbres' connection management strategy, comparing it with two other strategies: a full connection strategy (Fully) and a random partial connection strategy (Partially). The analysis was performed under varying levels of imbalance and SLA, as depicted in Fig. 12(b). Compared to the full connection strategy, the partial connection strategy required, on average, 42.3% more containers to meet the SLA requirements. The performance degradation of the partial connection strategy was further exacerbated

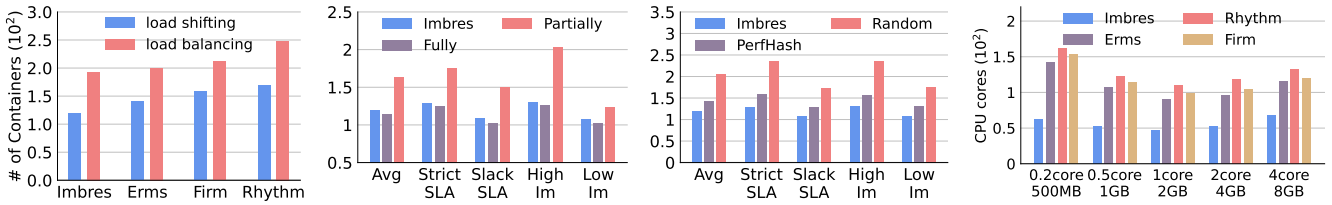


Figure 12. (a) (b) (c): Performance enhancement from Imbres' load shifting, connection management, and identification of critical microservices, respectively. (d): Resource allocation under varying configurations of microservice containers.

in high imbalance scenarios, resulting in a 61.8% increase in resource allocation.

In contrast, Imbres employs a highly efficient connection management strategy that achieves performance comparable to the full connection strategy, with an increase in resource allocation of less than 7%. This slight increase is due to the benefits of full connectivity, which enables effective load shifting among containers with minimal overhead in small-scale deployments, such as the 85 downstream containers in this case, resulting in better performance balance. However, as the scale of the deployment increases, the overhead of full connections becomes more significant. The benefits of performance balance are outweighed by the additional system overhead of full connections. Specifically, when the number of downstream microservices increases from 85 to 200, the E2E latency with full connections degrades by 10%, as shown in Fig. 11, thereby outweighing the 7% performance advantage compared to Imbres. As a result, the full connection strategy ultimately yields inferior performance compared to Imbres in larger-scale settings. These results highlight the effectiveness of Imbres' connection adjustments, particularly in production environments.

Why identify critical microservices? In this part, we emphasize the importance of accurately identifying the critical microservices when *Per-Microservice Scaling* allocates containers. Recall that the score of a microservice is determined by the product of its performance hash value, g_i, and the load imbalance index, α. To assess the efficacy of this approach, we implemented two additional schemes. One scheme evaluates microservices based solely on the values of the performance hash table (PerfHash). The other scheme randomly selects a microservice for resource scaling (Random). Figure 12(c) illustrates the average resource allocation by each system under different settings. Utilizing a performance hash table for resource scaling, as opposed to randomly selecting microservices, results in a notable 32% resource savings. This underscores the effectiveness of utilizing the performance hash table in identifying critical microservices. Furthermore, Imbres achieves an additional 13% reduction in resource scaling compared to schemes that only rely on the performance hash table. The primary reason behind this improvement is that the load imbalance index α accurately estimates which microservices are sensitive to interference imbalance, helping mitigate its impact.

Performance under various container configurations. To assess the impact of container configurations on the effectiveness of Imbres, we compared it with other baseline schemes across varying configurations, ranging from 0.2 CPU cores with 500 MB of memory to 4 CPU cores with 8 GB of memory per container. This evaluation was conducted using the Train Ticket application in a cluster comprising 16 hosts. As shown in Fig. 12(d), when the container configuration is increased from 0.2 CPU cores with 500 MB of memory to 0.5 CPU cores with 1 GB of memory, the reduction in resource allocation of Imbres relative to baselines declines from 62% to 54%. This decline is due to the number of containers for downstream microservices dropping to fewer than 40 within this limited-scale cluster. In such scenarios, the advantages of connection management diminish compared to random partial connection, as an upstream container is already connected to most downstream containers.

When the container configurations are further increased to 4 CPU cores and 8 GB of memory, the number of containers for downstream microservices falls below 10. In this case, full connection leads to a resource overhead of less than 4% compared to the connection management used in Imbres, as quantified in § 6.3. Moreover, in these scenarios, the performance of microservices becomes less susceptible to resource interference. Nevertheless, Imbres still achieves a 45% reduction in resource allocation compared to all baseline schemes. Additionally, we conducted trace-driven simulations using data from Alibaba [1], simulating a cluster deploying thousands of hosts. Each container was configured with 4 CPU cores and 8 GB of memory, aligning with production specifications. The results indicate that, for the most complex application, where each microservice is deployed with hundreds of containers, Imbres achieves a reduction of up to 72% compared to all baselines. These findings underscore Imbres' effectiveness in environments with larger container configurations and greater scales.

6.5 Convergence Performance

We deployed the Social Network application [18] on Amazon EC2 to evaluate the convergence efficiency of Imbres.

Convergence of load shifting process. We first disabled the resource scaling and initiated the load shifting under static workloads. Fig. 13(a) compares the E2E latency of the application under dynamic load control (LS) against

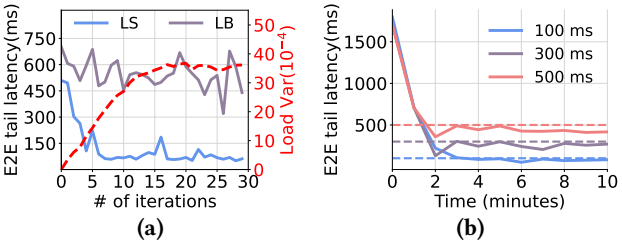


Figure 13. (a) 95th-percentile E2E latency and load variance during iterations of load shifting. LS denotes load shifting and LB denotes load balancing. (b) Tail Latency under Imbres.

latency under load balancing (LB). The E2E latency rapidly decreases to a near-optimal value within 6 iterations (30 seconds), demonstrating the rapid convergence of Imbres' load shifting process. We also tracked the load variance of microservices. As depicted by red lines in Fig. 13(a), the load variance converges rapidly within 12 iterations (1 minute). After reaching the optimal value, the load variance stabilizes with minor fluctuations.

Convergence of resource scaling process. We demonstrate the convergence efficiency when the load control and resource scaling concurrently iterate. To achieve this, we initialized the total number of containers for the whole application to one-fourth of the optimal configuration when the SLA is set to 100ms. We generated workloads to assess the efficiency of Imbres with both controls enabled under different SLA settings, injecting the highest interference imbalance. As illustrated in Fig. 13(b), the E2E latency of the application converges with minor fluctuations within two minutes under different SLA settings. These results affirm the optimal coordination between the load control and resource scaling and the high convergence efficiency of the three-tiered control of Imbres, enabling it to promptly react to system fluctuations and find near-optimal resource configurations.

6.6 Sensitivity Analysis

Here, we analyze Imbres' sensitivity to the hyperparameters involved in its design. We conducted experiments for all four applications, with each generating dynamic workloads for 30 minutes using various hyperparameters. As illustrated in Fig. 14, the thin red bars indicate the full range of observed values across the four applications for each parameter setting, with the upper and lower bounds representing the highest and lowest recorded values, respectively. Meanwhile, the thicker bar represents the average result across all applications for each parameter setting.

Optimal algorithm hyperparameters. We assessed the impact of different parameter values in Imbres' algorithm. As depicted in Fig. 14(a), resource allocation initially decreases with increasing learning rates used in Imbres, denoted as η in *Load Shifting* and η_1, η_2 in *Per Application Scaling*, as these rates accelerate the convergence of Imbres' algorithm. However, excessively high learning rates can cause the algorithm

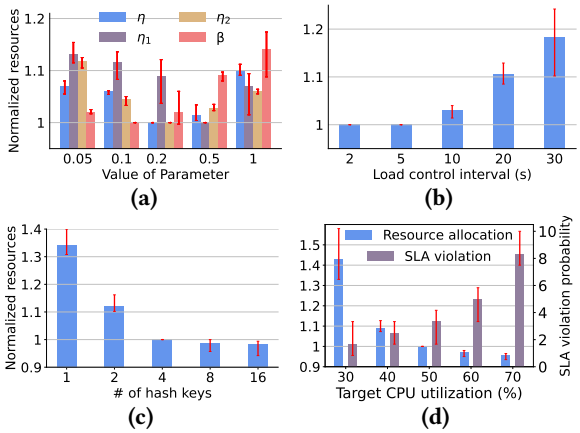


Figure 14. Resource allocation under different parameter settings. (a) Imbres' algorithmic parameter. (b) Load control interval. (c) Number of hash keys. (d) Target CPU utilization.

to converge to suboptimal solutions, leading to increased resource allocation. Additionally, in the Per Application Scaling module, the parameter β serves as the threshold for switching from first-order gradient descent to Newton's method. Setting β to a value greater than 0.1 may prevent the adoption of the more efficient Newton's method, delaying convergence to optimal scaling solutions and thereby increasing overall resource allocation. However, a smaller value, such as 0.05, can lead to resource overprovisioning due to gradient explosion. Despite these variations, the overall resource allocation differs by less than 15% across all parameter settings, highlighting the robustness of Imbres. Furthermore, for all parameter settings, the difference between the highest and lowest result is less than 15% of the average value, indicating that Imbres is highly adaptable across different applications.

Impact of load control interval. Extending the load control interval increases the convergence time of Imbres' load shifting process. It leads to overprovisioning of resources to prevent SLA violations caused by imbalances in performance among microservice containers before load shifting converges. Consequently, this requires additional iterations for the resource scaling module to achieve convergence. Conversely, a shorter load control interval is constrained by the trace collection efficiency of the tracing system Jaeger [2]. As shown in Fig. 14(b), when the load control interval is extended from 5 to 30 seconds, overall resource allocation increases by 18%. However, reducing the load control interval from 5 to 2 seconds provides only marginal benefits in resource efficiency. Furthermore, with an increasing load control interval, the variance also increases, as shown in Fig. 14(b), reaching up to 14.8% of the average value. To balance the trace collection overhead and resource efficiency, we set the default load control interval to 5 seconds. The optimal interval remains consistent across all applications.

Selection of hash key groups. The number of hash key groups is closely linked to the prediction accuracy of microservice performance. Increasing the number of hash

keys improves prediction accuracy, facilitating better scaling decisions, but it also incurs higher storage overhead. As shown in Fig. 14(c), resource efficiency significantly improves when the number of hash keys is increased from two to four. However, further increasing the number to sixteen results in over 256 times higher storage consumption, with only a marginal resource efficiency gain of less than 3%. To achieve an optimal balance, Imbres selects four key groups for the performance hash table by default. For different numbers of hash keys, the variation across applications remains within 8% as shown in Fig. 14(c). This suggests that the number of hash keys is not strongly tied to specific applications.

Determining target CPU utilization. Imbres uses the target CPU utilization of containers to determine the maximum load that individual microservice containers can support. A higher target encourages more aggressive resource allocation, enhancing resource efficiency but resulting in increased rates of SLO violations. As shown in Fig. 14(d), increasing the target CPU utilization by 10% reduces resource allocation by 12% on average, but simultaneously raises the SLA violation rate by 79%. To strike a balance between resource savings and SLA violations, Imbres sets the target CPU utilization at 50%. This choice is further supported by the observation that increasing the target from 50% to 70% increases the SLO violation probability by 1.6×, while improving resource efficiency by only 4.2%. This configuration aligns with the CPU resource utilization patterns observed in production clusters [1, 50].

7 Discussion

In this section, we discuss how Imbres can be extended to achieve broader applicability in practical scenarios.

LLC contention. LLC contention is another source of resource interference that impacts the cache hit rate of microservices. We conducted experiments to assess the effect of LLC contention on microservice latency, and the results indicate that latency can increase by 2.3× when the LLC miss rate exceeds 60% on NUMA nodes in extreme cases. Currently, LLC contention has not been incorporated into Imbres' design. However, by utilizing Linux's perf tool [6], LLC contention can be easily integrated into the performance hash table as an additional source of interference.

The limit of Imbres' improvement. In extreme scenarios where resource interference differences across hosts can reach up to 80%, the experimental results demonstrate our load-shifting strategy can reduce resource allocation by as much as 80% compared to all baseline approaches within a 16-host cluster. This underscores the significant potential for enhancing resource efficiency through load shifting in production clusters with thousands of hosts.

8 Related Work

Load management. Load management, especially in the realm of online services, stands as a prominent and actively researched area [7, 13, 14, 27, 40, 45, 51, 54, 59]. Prequal is a specialized load balancer designed for distributed, multi-tenant systems [54]. Its primary objective is to minimize real-time request latency, even in the face of disparate server capacities and fluctuating antagonistic workloads. DAGOR and Breakwater [13, 59] address microservices overload through real-time monitoring and coordinated admission control. Kraken [51] conducts load tests by shifting live traffic across data centers, utilizing real-time data to identify and rectify resource utilization bottlenecks. Despite the advancements, these systems have not yet explored the integration of load shifting and connection management to optimize microservice architectures.

Microservice scaling. Numerous efforts have been dedicated to optimizing resource management for microservices [10, 12, 15, 17, 19, 22, 33, 35–38, 41, 43, 47, 55–58]. GrandSLAm [22] quantifies the contribution of each microservice to E2E latency for further resource allocation. DeepRest [15] incorporates a graph neural network for resource scaling. Sinan [57] estimates the E2E latency of a microservices application under different resource configurations using deep learning algorithms. Nodens [47] features a traffic-based load monitor, load updater, and query drainer to optimize network bandwidth for quick QoS recovery, efficiently managing resources for microservices. One big limitation of these approaches is their dependency on comprehensive offline profiling, making them struggle to adapt to the continuously evolving nature of microservices under dynamic resource interference environments.

9 Conclusion

This paper marks a pioneering effort in concurrently shifting microservice load and scaling resources amid highly dynamic and imbalanced resource interference. A distinct feature of our system is its exclusive reliance on online gradients derived purely from latency feedback to guide the optimization process, eliminating the need for any offline profiling. Furthermore, the system utilizes performance indicators generated during load shifting to effectively identify critical microservices for resource scaling.

Acknowledgements

We sincerely thank the anonymous ASPLOS'25 reviewers, and our shepherd, Dr. Carl Waldspurger, for their insightful suggestions. This work is supported in part by the National Key Research and Development (R&D) Plan under Grant 2022YFB4500004, the Science and Technology Development Fund of Macau (0024/2022/A1, 0071/2023/ITP2, 0081/2022/A2, and 0123/2022/AFJ), the Guangdong Basic and Applied Basic Research Foundation under Grant 2023B1515020120, as well as the Multi-Year Research Grant of University of Macau (MYRG-GRG2023-00019-FST-UMDF, MYRG-GRG2024-00255-FST-UMDF).

References

[1] 2022. Alibaba Microservices Cluster Traces. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022.

[2] 2023. Jaeger. https://jaegertracing.io/.

[3] 2023. Prometheus. https://prometheus.io/.

[4] 2024. AWS EC2. https://aws.amazon.com/ec2/.

[5] 2024. CNCF:. https://www.cncf.io/.

[6] 2024. Linux perf tool. https://perfwiki.github.io/main/.

[7] Swarup Acharya, Michael Franklin, and Stanley Zdonik. 1997. Balancing push and pull for data broadcast. In *Proceedings of SIGMOD*.

[8] Kubernetes's Horizontal Pod Autoscaling. 2024. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[9] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take It to the Limit: Peak Prediction-Driven Resource Overcommitment in Datacenters. In *Proceedings of EuroSys*.

[10] 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*.

[11] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. 2023. Remote Procedure Call as a Managed System Service. In *Proceedings of NSDI*.

[12] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of ASPLOS*.

[13] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for μs-scale RPCs with Breakwater. In *Proceedings of OSDI*.

[14] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, et al. 2019. Taiji: managing global user traffic for large-scale internet services at the edge. In *Proceedings of SOSP*.

[15] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. 2022. DeepRest: Deep Resource Estimation for Interactive Microservices. In *Proceedings of EuroSys*.

[16] Christina Delimitrou and Christos Kozyrakis. 2013. IBench: Quantifying interference for datacenter applications. In *Proceedings of IISWC*.

[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] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-Driven Autoscaling for Microservices. In *Proceedings of ICDCS*.

[20] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of ASPLOS*.

[21] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. 2023. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *Proceedings of ATC*.

[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] Carl T Kelley. 2003. *Solving nonlinear equations with Newton's method*. SIAM.

[24] Leonard Kleinrock. 1975. *Queueing systems. Volume 1: theory*.

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

[26] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of ASPLOS*.

[27] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. 2021. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of SOSP*.

[28] Chieh-Jan Mike Liang, Zilin Fang, Yuqing Xie, Fan Yang, Zhao Lucis Li, Li Lyna Zhang, Mao Yang, and Lidong Zhou. 2023. On Modular Learning of Distributed Systems for Predicting {End-to-End} Latency. In *Proceedings of NSDI*.

[29] Chenyu Lin, Shutian Luo, and Huanle Xu. 2023. Exploring Imbalances among Microservice Containers in Large Cloud Platforms. In *Proceedings of IEEE ISPA*.

[30] Chengzhi Lu, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2023. Understanding and Optimizing Workloads for Unified Resource Management in Large Cloud Platforms. In *Proceedings of Eurosys*.

[31] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on alibaba cluster trace. In *Proceedings of Big Data*.

[32] 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*.

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

[34] 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* (2022).

[35] 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*.

[36] 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*.

[37] 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*.

[38] 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*.

[39] 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*.

[40] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. 2013. Ananta: Cloud scale load balancing. In *Proceedings of SIGCOMM*.

[41] 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*.

[42] Redis. 2024. https://redis.io/.

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

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

[45] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. {ServiceRouter}: Hyperscale and Minimal Cost Service Mesh at Meta. In *Proceedings of OSDI*.

[46] Shai Shalev-Shwartz et al. 2012. Online learning and online convex optimization. *Foundations and Trends® in Machine Learning* 4, 2 (2012), 107–194.

[47] Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo. 2023. Nodens: Enabling Resource Efficient and Fast {QoS} Recovery of Dynamic Microservice Applications in Datacenters. In *Proceedings of USENIX ATC*.

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

[49] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, and Jonathan Kaldor, et al. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of OSDI*.

[50] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of EuroSys*. 1–14.

[51] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *Proceedings of OSDI*.

[52] Peter Ward, Paul Wankadia, and Kavita Guliani. 2022. Reinventing Backend Subsetting at Google: Designing an algorithm with reduced connection churn that could replace deterministic subsetting. *Queue* (2022).

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

[54] Bartek Wydrowski, Robert Kleinberg, Stephen M Rumble, and Aaron Archer. 2024. Load is not what you should balance: Introducing Prequal. In *Proceedings of NSDI*.

[55] 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*.

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

[57] 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*.

[58] 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*.

[59] 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*.

[60] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Software Eng.* (2021).