2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00044

Grad: Intelligent Microservice Scaling by Harnessing Resource Fungibility

Liao Chen†, Chenyu Lin†, Shutian Luo‡, Huanle Xu†*, Chengzhong Xu†
University of Macau†, Yale University‡
chenliaao.16, mc14889@connect.um.edu.mo, shutian.luo@yale.edu, huanlexu, czxu@um.edu.mo

***Abstract*—Microservice applications are commonly deployed alongside other services to enhance resource utilization. However, this practice also leads to notable resource contention. While existing studies primarily focus on scaling critical microservices responsible for performance degradation to mitigate violations of SLAs regarding end-to-end latency in highly interfered environments, they often overlook the potential advantages of scaling non-critical microservices for optimized resource efficiency.**

In this paper, we introduce Grad, an intelligent microservice scaling framework by harnessing resource fungibility between critical and non-critical microservices. Addressing the challenges posed by the dynamic nature of resource fungibility during scaling, Grad incorporates three key components. First, Grad employs a modular learning approach to profile individual microservice latency in relation to environmental conditions. Utilizing gradient extracts from this profile, Grad designs a scalable optimization module to dynamically select the optimal set of microservices for scaling. To rapidly mitigate SLA violations, Grad also deploys an accurate end-to-end latency predictor, serving as an simulator to obtain real-time feedback. We evaluate Grad in our cluster using real microservice benchmarks and production traces, demonstrating its ability to reduce resource usage by 49.1% and lower the probability of SLA violations by 3.7× when compared to state-of-the-art solutions.

I. Introduction

Nowadays, microservice architecture has gained immense popularity in building a wide range of latency-sensitive online services with exceptional flexibility [1], [22], [38], [39], [42], [52]. Unlike monolithic architectures, this new approach breaks down complex applications into multiple loosely-coupled components, simplifying their administration, maintenance, and updates. Consequently, when confronted with heavy loads or resource bottlenecks, microservice architecture enables independent and fine-grained resource scaling of individual components, as opposed to the entire application [7], [8], [19], [21], [29], [34], [47], [49].

Meanwhile, today's cloud service providers often choose to co-locate microservice applications with other online services or batch processing jobs within the same cluster, aiming to enhance resource utilization [6], [11], [26], [40], [51]. However, this shared environment often results in considerable resource contention, encompassing CPU, memory, LLC, and network [8], [35], among individual microservices and other co-located tasks, thereby substantially impacting the end-to-end (E2E) latency of the entire microservice application [35]. Analysis of traces from production clusters reveals that resource interference is highly dynamic over time and varies significantly across different physical machines [23].

To optimize microservice scaling in complex, interference-prone environments and ensure adherence to service level agreement (SLA) requirements regarding tail E2E latency, current research focuses on two main approaches [28], [35]. The first approach constructs explicit linear models to quantify the relationship between interference and performance [28]. Although this approach can coordinate resource scaling across multiple microservices globally, its oversimplified nature restricts scaling exclusively to horizontal scaling alone, leaving per-container resource configurations unchanged. This absence of scaling up capability leads to exceedingly low resource efficiency within a dynamic environment. The second methodology relies extensively on machine learning (ML) techniques to identify critical microservices causing SLA violations due to resource contention. It then employs reinforcement techniques to allocate additional resources to these critical microservices based on real E2E latency feedback, achieved through a combination of scaling out and scaling up [35]. Nevertheless, concentrating solely on scaling a limited set of critical microservices can result in suboptimal resource efficiency. Furthermore, the reinforcement approach requires real-time feedback, taking seconds to mitigate SLA violations.

Our analysis of microservice benchmarks [15] reveals the fungibility of various computing resources between critical and non-critical microservices during scaling within interfered environments. In contrast to critical microservices, whose performance is highly susceptible to interference, certain non-critical microservices exhibit high sensitivity to resource availability. In many cases, leveraging resource fungibility—the ability to substitute or reallocate resources like CPU, memory, I/O, among others across both critical and non-critical microservices—can lead to more balanced and efficient scaling decisions. Consequently, focusing solely on scaling critical microservices overlooks the potential for optimizing resource efficiency. As demonstrated in § II-B, scaling microservices closely interdependent with critical ones or those more sensitive to resource allocation can reduce tail E2E latency by 25% compared to scaling critical microservices alone. As a result, it is essential to harness resource fungibility during scaling through effective resource trading among microservices within the same application.

While the concept of resource fungibility offers great potential in enhancing resource efficiency, achieving optimal resource trading within complex microservice applications

*Huanle Xu is the corresponding author.

979-8-3315-0647-6/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00044

during scaling introduces several new challenges. The primary challenge arises from the dynamic nature of selecting the suitable microservice for trading, a task complicated by its strong causal relationship with previous selections. Consequently, the sequencing of microservice selection for scaling becomes crucial. This challenge is further compounded by the dynamic sensitivity of microservice latency to different resource types during scaling, which varies widely based on container resource utilization, load, and interference conditions. Hence, there arises a necessity for a dynamic trading policy capable of swiftly adapting to environmental changes.

To effectively address these challenges and fully leverage resource fungibility, we have developed Grad, an intelligent microservice resource management system designed to promptly mitigate SLA violations while minimizing resource usage in shared clusters. Grad embodies three fundamental design principles. First, it adopts a modular learning approach [21] to model the latency of each microservice. This model considers container resource utilization (such as CPU and memory), load, and diverse resource interferences (such as cache, memory bandwidth, I/O, and network interferences) on the machine where the microservice resides. Powered by shallow neural networks, this modular framework is lightweight and adept at capturing environmental dynamics in a scalable manner. Furthermore, it facilitates both horizontal and vertical scaling by harnessing the fungibility among various resource types. Second, Grad integrates a fine-grained optimization module to dynamically select the appropriate microservice and the associated resource type for scaling when the load fluctuates or resource interferences vary. This module utilizes gradients extracted from the neural networks, effectively capturing the causal relationship between successive selections. Third, to rapidly mitigate SLA violation during scaling, Grad incorporates an E2E latency predictor using self-attention [43]. Given that the microservice selection process may entail multiple steps, this predictor functions as an ideal simulator, offering real-time feedback for each scaling decision without the need for immediate deployment within the cluster.

We have implemented a prototype of Grad on the Kubernetes platform [13], [20] and deployed it in our local cluster comprising ten servers, each with 104 cores. To evaluate Grad, we conduct extensive experiments utilizing the widely adopted DeathStarBench [15] and TrainTicket [53] benchmarks, as well as large-scale trace-driven simulations based on Alibaba traces [2]. The results demonstrate that Grad achieves a prediction accuracy of nearly 90% for both individual microservice latency and tail E2E latency. Importantly, experimental results highlight Grad's ability to reduce overall resource usage by up to 49.1% and decrease the probability of SLA violations by 3.7 times compared to state-of-the-art approaches. In summary, this paper makes the following contributions:

- **Comprehensive analysis of resource fungibility**. We demonstrate the presence of resource fungibility between critical and non-critical microservices within heavily interfered environments. We also analyze how this fungibility is profoundly dynamic, depending on the sequence of microservice scaling and diverse environmental factors.
- **Design of optimal resource scaling mechanism**. We introduce a novel optimization method grounded in gradient computation and Taylor approximation to dynamically pinpoint the optimal microservices for resource trading with low computational overhead.
- **Implementation of system with rapid response**. Our microservice scaling system implementation is capable of promptly addressing issues within 130 millseconds in instances of SLA violations, representing an order of magnitude improvement over existing implementations.

II. Background and Motivation

A. Microservices in Shared Clusters

In a production cluster, numerous applications are hosted across thousands of machines, each typically providing a variety of online services customized to meet user requests. These services usually consist of multiple microservices, each deployed with hundreds of containers with uniform configurations [27]. Ordinarily, a user request is directed to an entry microservice, such as Nginx [15], which then initiates a series of invocations among dependent microservices, either sequentially or in parallel. The E2E latency of a request is defined as the duration from the moment a user sends a request to the time it receives the corresponding reply [28].

Microservice containers often share computing resources with other services across the same machines [26]. Although this shared environment can boost resource utilization by over 50% [26], it concurrently brings about significant resource contention [27], [28], [35]. Analyzing traces from Alibaba clusters reveals that resource interference can result in over a fourfold increase in latency for individual microservices [23]. As a result, it is crucial for microservice frameworks to effectively scale resources within this highly interfered environment, ensuring they can handle dynamic request arrivals while meeting the SLA requirements for tail E2E latency.

B. Opportunity: Leveraging Resource Fungibility between Critical and Non-critical Microservices

A critical microservice often resides on the longest path in the request's execution graph and is distinguished by significant latency variations across different percentiles, such as the 50th percentile versus the 95th percentile. As defined by [35], the variability of a critical microservice contributes the most to the overall variability of the E2E latency within an application. Critical microservices are highly susceptible to notable performance degradation due to resource contention, leading to SLA violations. Consequently, previous studies have achieved significant advancements in employing ML techniques to identify critical microservices for resource scaling, with the aim of reducing E2E latency. Nevertheless, our observations indicate that scaling critical microservices may not result in the most resource-efficient solution.

To illustrate this point, we conducted an experiment utilizing the `ComposePost` service from Social-Network application featured in DeathStarBench [15]. The `ComposePost` service

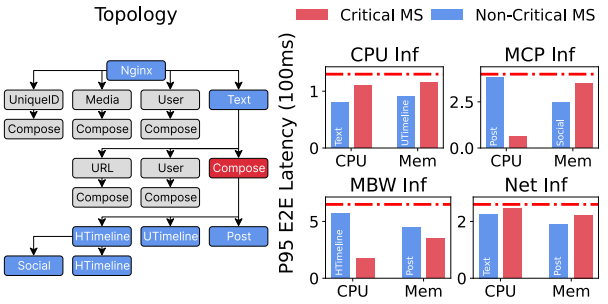


Fig. 1: Benefit of leveraging resource fungibility. The red dotted line represents the initial tail E2E latency before scaling, while the red bars illustrate the tail E2E latency after scaling for the critical microservice experiencing resource interference (Inf). The blue bars indicate the optimal performance attained by allocating more resources to non-critical microservices.

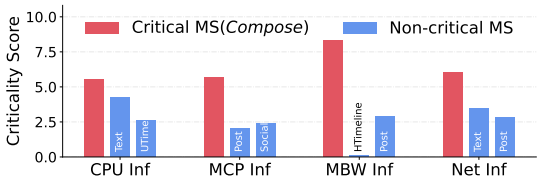


Fig. 2: Criticality score quantified using latency variability at both the microservice and application levels.

comprises twelve stateless microservices, as illustrated in Fig. 1. Our emphasis lay on the `Compose` microservice, which is highly susceptible to resource interference. We introduced various interferences to the hosting machine of `Compose` under the same load, thus highlighting its significance as a critical microservice in violating the SLA requirement of tail E2E latency. Precisely, we introduced interferences in CPU, memory capacity (MCP), memory bandwidth (MBW), and network (Net), employing a mechanism akin to that explored in Firm [35], and utilizing the IBench tool [12] and Iperf3 [3] for interference injection. In order to showcase the consistency of critical microservices under varying interference, we also quantify each microservice's correlation with tail E2E latency by measuring the variability in their individual latency distribution [35]. As depicted in Fig. 2, it is evident that the `Compose` consistently achieves the highest values across various scenarios.

We augmented identical CPU or memory resources across all microservices within the `ComposePost` service sequentially, while measuring the resulting tail E2E latency for each scenario. As shown in Fig. 1, during CPU, memory capacity, and network interference impacting the `Compose` microservice, allocating CPU or memory resources to other non-critical microservices such as `Text`, `Social`, `Post`, and `Utimeline` significantly reduces the tail E2E latency (up to 25%), compared to a strategy that only scales `Compose`.

The reason behind this phenomenon is the close dependency of these microservices on `Compose`, or their high sensitivity to resource availability, offering ample opportunities for trading resources among them and `Compose` to enhance efficiency. This observation highlights two levels of resource

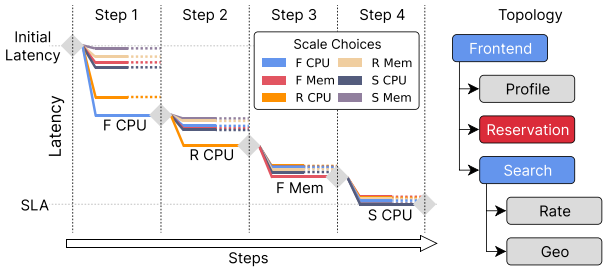


Fig. 3: E2E latency of resource scaling across different steps. The lines indicate the changes in latency under different scaling decisions, with the solid line representing the optimal choice made at each step. In the graph, F represents `Frontend`, R represents `Reservation`, and S represents `Search`. CPU and Mem respectively denote the allocation of CPU and memory.

fungibility. First, a portion of the resources allocated to critical microservices to address SLA violations can be reallocated to non-critical microservices to improve efficiency. Second, CPU and memory resources exhibit fungibility, meaning that even if a critical microservice is experiencing memory interference, allocating CPU resource to it can more effectively mitigate performance degradation. For instance, when `Compose` experiences CPU or memory contention, augmenting CPU or memory to it cannot effectively alleviate the bottleneck, as opposed to scaling an equivalent amount of resources for its upstream microservice `Text` or another sensitive microservice `Social`. Conversely, when `Compose` faces memory bandwidth interference, trading resources between `Compose` and any other microservices leads to a degradation of E2E latency.

C. Challenge: Resource Fungibility Is Highly Dynamic

Achieving optimal resource trading is challenging, primarily because resource fungibility is inherently dynamic.

The primary hurdle lies in selecting the appropriate microservice for trading, a dynamic task complicated by its strong causal relationship with previous selections. Consequently, the sequence in which microservices are selected for scaling becomes pivotal in enhancing resource efficiency. To validate this perspective, we conducted another experiment utilizing a simpler service called `Search` from the HotelReservation application in DeathStarBench. During this experiment, we introduced 40% CPU and 35% memory capacity interference[1] on the machines hosting the `Reservation` microservice under consistent load. Furthermore, we augmented resources separately with a granularity of 0.2 CPU core or 100MB memory to each microservice within the `Search` service, as illustrated in Fig. 3. The optimal fungible microservice, paired with the critical microservice `Reservation`, was identified as the one that resulted in the smallest E2E latency. We iterated this experiment multiple times, with each step based on the optimal scaling result from the previous step.

[1]The quantification of interference, such as 35% memory capacity, indicates that a node hosting a microservice has 35% of its memory occupied by other tasks. This quantification applies similarly to the other three types of interferences, all based on time-division multiplexing.

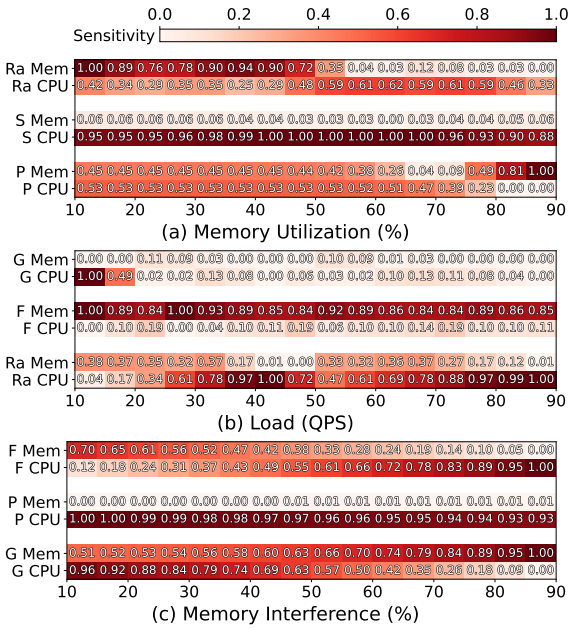


Fig. 4: The influence of different factors on microservice sensitivity to latency, where darker shades indicate higher sensitivity. The values in the heatmap signify the absolute latency reductions achieved by augmenting corresponding resources. Ra, S, P, G, F respectively represent the `Rate`, `Search`, `Profile`, `Geo`, and `Frontend` microservices.

As depicted in Fig. 3, the optimal resource trading decision, which involves determining both the fungible microservice to scale and the type of resource to augment, varies across all steps. In the initial phase, augmenting CPU resources for `Frontend` rather than the critical microservice `Reservation` proves more advantageous in reducing end-to-end latency by up to 30%. However, augmenting CPU resources directly for `Reservation` emerges as the most efficient approach following the initial scaling outcomes, given the updated status of `Frontend` after resource augmentation. Furthermore, in subsequent stages, enhancing memory resources for `Frontend` and CPU resources for `Search` becomes increasingly beneficial. This suggests that selecting just one microservice to trade with the critical microservice undermines resource efficiency.

The second challenge arises from the dynamic sensitivity of microservices in responding to resource scaling, which is influenced by factors such as container resource utilization, load, and interference. This characteristic amplifies the difficulty of selecting the right microservices and their associated resource type for resource trading under evolving environmental conditions. To elucidate this notion, we embarked on a series of new experiments centered around the `Search` service. Through these experiments, we deliberately manipulated resource utilization of microservice containers, loads, and interference to examine how individual microservices respond. To achieve this, we systematically adjusted one factor within a normalized range from 10% to 90%, while maintaining all other factors influencing microservice performance constant. Subsequently, we incrementally allocated CPU (in 0.2 core increments) or memory (in 100 MB increments) to each microservice, measuring the resulting latency of each. The greater the absolute reduction in latency, the higher the sensitivity of the microservice to scaled resources.

As depicted in Fig. 4(a), the three microservices (`Rate`, `Search`, `Profile`) exhibited distinct dynamic sensitivity trends when memory utilization changes. For microservice `Rate`, augmenting memory outperformed increasing CPU in latency reduction when memory utilization of containers was below 50%, whereas beyond 50%, augmenting CPU yielded greater performance improvements. Meanwhile, `Search` microservice consistently demonstrated that enhancing CPU outperformed increasing memory, irrespective of changes in memory utilization. As for `Profile`, when memory utilization was below 75%, increasing CPU proved to be the superior choice, while beyond 75%, the microservice became more sensitive to memory. Similarly, we investigated the sensitivity changes in `Geo`, `Frontend`, and `Rate`, under varying QPS load (100 to 1000 requests/second) with resource utilization of containers and interference fixed. As illustrated in Fig. 4(b), these three microservices also exhibited dynamic sensitivity trends on different resource types with the change of loads. Finally, we varied the resource contention on machines where microservices reside from low to high. As depicted in Fig. 4(c), when `Frontend` faces high interference in memory capacity, increasing CPU can yield better performance improvements than augmenting memory. These results suggests a dynamic fungibility among different resource types exists for scaling, highlighting that placing excessive emphasis on contended resources may be less effective. However, it is crucial to note that in production environments, the complexity of scaling further amplifies as multiple factors interact simultaneously.

III. Overview of Grad

This section outlines the framework of Grad, designed as a microservice resource management solution adept at orchestrating rapid microservice scaling in response to environmental fluctuations by harnessing resource fungibility.

A. Key Design Ideas

I_1 : **Quantifying microservice sensitivity through modular learning**. Grad first tackles the latter challenge outlined in § II-C by employing a modular learning approach to continuously estimate the dynamic sensitivity of microservice scaling to environmental changes, including factors such as container resource utilization, load variations, and various resource contention scenarios. This scalable learning process is driven by a Latency Profiling module, which systematically profiles microservice latency in relation to these factors.

I_2 : **Addressing causality through gradient computation**. Addressing the additional challenge outlined in § II-C, Grad leverages Taylor approximation and gradient computation to explicitly capture causal relationship between successive selections of microservices for resource trading. Given the iterative

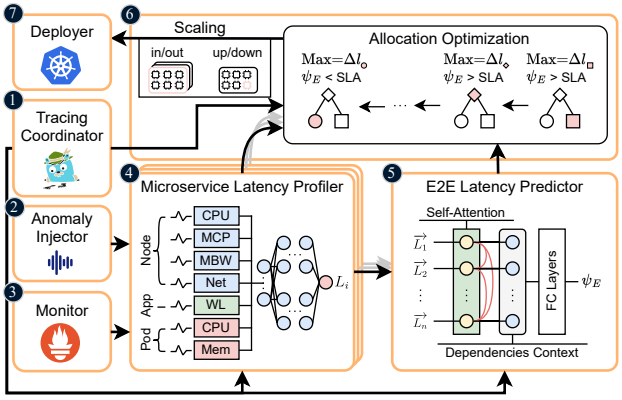


Fig. 5: The system architecture of Grad.

nature of the fine-grained selection process, Grad also relies on simulated real-time feedback from E2E Latency prediction to guide the scaling process.

B. System Architecture

Grad deploys a *Tracing Coordinator* (❶ in Fig. 5) and a *Monitor* (❸ in Fig. 5) atop Jaeger [4] and Prometheus [5], respectively. The *Tracing Coordinator* captures execution paths dynamically for all online services and extracts individual microservice latencies for all deployed microservices, along with tail E2E latency, drawing from historical traces. Simultaneously, the *Monitor* primarily gathers operating system-related metrics, including resource utilization/usage for each container and the resource status of their respective physical hosts. This data assists in quantifying dynamic resource contention within the cluster. Furthermore, the *Monitor* is responsible for collecting real-time load data during runtime.

The *Microservice Latency Profiling* module (❹ in Fig. 5) utilizes data sourced from the *Tracing Coordinator* and *Monitor* to gather samples of microservice latency under diverse conditions. These samples are subsequently employed to train a shallow neural network for each microservice. Meanwhile, the *E2E Latency Predictor* module (❺ in Fig. 5) captures the E2E latency of each request, facilitating the learning of tail E2E latency based on the latency of individual microservices. To collect sufficient training data for profiling, Grad also integrates an *Anomaly Injector* (❷ in Fig. 5), introducing diverse interferences targeting the machine hosting the microservices.

The *Allocation Optimization* module (❻ in Fig. 5) is the core component of Grad. This module leverages offline profiles to formulate resource scaling decisions for each microservice across varying conditions. Specifically, it gathers explicit knowledge into the sensitivity of microservice latency to resource scaling and devises an optimal scaling strategy based on the current state. Moreover, it utilizes the *E2E Latency Predictor* to assess the disparity between the estimated tail E2E latency of the generated scaling strategy and the SLA, determining the necessity for further scaling. Scaling actions are executed on the underlying Kubernetes cluster through the *Deployment* module (❼ in Fig. 5).

IV. Characterizing Dynamic Resource Fungibility

A. Modularized Learning of Microservice Latency

Grad adopts a modular learning approach to model the latency of individual microservices, contrasting with the monolithic models employed by Sinan [49], Graf [34], and Sage [14]. The primary rationale behind this choice is that these models integrate all features into a unified end-to-end learning framework, thereby constraining their scalability and facing challenges in effectively incorporating real-time interference conditions. Moreover, monolithic models cannot adapt to dynamic deployments where microservices are frequently replaced over time; retaining such models proves to be quite time-consuming [21].

Fluxion [21] similarly adopts a modularized approach to learn individual microservice latency, requiring each learning model to encompass all dependencies from downstream microservices. While explicitly modeling dependencies can enhance accuracy in subsequently predicting tail E2E latency, it concurrently restricts scalability in extending the input feature space to encompass various resource interference metrics—critical for Grad to leverage resource fungibility in optimizing scaling. To effectively address this limitation, the *Latency Profiling* module of Grad selectively eliminates dependencies when analyzing the latency of individual microservices. Specifically, this module computes the latency l_i of each microservice i by subtracting the response times of all its downstream microservices from its own response time [28]. The response time measures the duration between the arrival of a request and the dispatch of the corresponding response. Essential request timestamp data for this computation can be sourced from *Tracing Coordinator*. Notably, this response time, incorporating both queuing and execution time during request handling, serving as a comprehensive indicator of the resource pressure encountered by an individual microservice.

Feature Selection. The *Latency Profiling* module selects features from three different types. These include the resource utilization vector $\overrightarrow{u_i} = (u_i^c, u_i^m)$, where u_i^c and u_i^m represent the CPU and memory utilization of microservice containers, respectively. Since resource utilization metrics are unavailable during inference, *Latency Profiling* resorts to utilizing resource usage as a substitute of $\overrightarrow{u_i}$. Additionally, the features encompass the per-container load, denoted by ρ_i, which can be calculated using the formula $\rho_i = \Phi_i / n_i$, where Φ_i represents the microservice load and n_i is the number of containers deployed within microservice i. Crucially, the features comprehensively capture fine-grained resource interference on the machine where a microservice is deployed, represented by a vector $\overrightarrow{I_i} = \left(I_i^c, I_i^{mc}, I_i^{mb}, I_i^{nb}\right)$, quantifying the extend of contention on CPU (I_i^c), memory capacity (I_i^{mc}), memory bandwidth (I_i^{mb}), network bandwidth (I_i^{nb}), respectively. Given that the input features entail multiple value types with markedly differing ranges, which can impede effective learning in certain dimensions, the *Latency Profiling* module addresses this issue by performing max-min normalization on each feature dimension of $\overrightarrow{u_i}$, w_i, and $\overrightarrow{I_i}$.

Latency Profiling. The *Latency Profiling* module learns the relationship between the tail (e.g., 95th percentile) of microservice latency and the corresponding features through a function f_i for each microservice i, represented as:

$$l_i = f_i(\overrightarrow{u_i}, \rho_i, \overrightarrow{I_i}). \tag{1}$$

The profiling module employs a shallow neural network to approximate this function f_i through data fitting. Neural network offers a more comprehensive and manageable model structure compared to other widely-used boosting models, such as XGBoost [9]. Consequently, this concise structure facilitates Grad to explicitly quantify the sensitivity of microservice latency to resource scaling, as detailed in § V, while also achieving high profiling accuracy.

B. Gradient is All You Need

Quantifying the dynamic sensitivity of each microservice to resource scaling along the execution path within a microservice application is crucial for effectively leveraging resource fungibility. By employing first-order Taylor approximation on the latency function of each microservice, the *Allocation Optimization* module of Grad can estimate the sensitivity to both vertical and horizontal scaling. Specifically, during vertical scaling where a total amount of r (CPU or memory) resources is added to all containers of the selected microservice for resource trading, the decrease in microservice latency $\triangle_i^v(r)$ can be computed as follows:

$$\triangle_i^v(r) = f_i\Big(\overrightarrow{u_i}+\frac{\overrightarrow{r}}{n_i}, \rho_i, \overrightarrow{I_i}\Big) - f_i\big(\overrightarrow{u_i}, \rho_i, \overrightarrow{I_i}\big) \approx \frac{\partial f_i}{\partial \overrightarrow{u_i}} \cdot \frac{\overrightarrow{r}}{n_i}. \tag{2}$$

In the computation, both CPU and memory resources are normalized using the max-min normalized method. Additionally, this approximation vectorizes r to align with the resource usage vector $\overrightarrow{u_i}$ by adding a zero to the type of resource that remains unchanged.

In contrast, during horizontal scaling, where more containers are added to microservice i with the current container configuration specified by $\overrightarrow{\eta_i}$, the *Allocation Optimization* module measures the performance gain $\triangle_i^h$ as follows:

$$\begin{aligned}\triangle_i^h(r) &= f_i\Big(\overrightarrow{u_i}, \frac{\Phi_i}{n_i + r/\eta_i}, \overrightarrow{I_i}\Big) - f_i\big(\overrightarrow{u_i}, \rho_i, \overrightarrow{I_i}\big) \\ &\approx \frac{\partial f_i}{\partial \rho_i} \cdot \Big(\frac{\Phi_i}{n_i + r/\eta_i} - \frac{\Phi_i}{n_i}\Big).\end{aligned} \tag{3}$$

For horizontal scaling, the *Allocation Optimization* module consolidates CPU and memory resources to capitalize on the fungibility between these two types of resources, contingent upon the capacity of the entire cluster. For instance, if the cluster contains 2000 CPU cores and 8000 GB memory, then 2 cores are treated as equivalent to 8 GB memory. The resource vector $\overrightarrow{\eta_i}$ is accordingly consolidated to generate a scalar value η_i. Consequently, the *Allocation Optimization* module determines the best scaling approach for microservice i by directly comparing between Eq. (2) and Eq. (3).

Calculating Gradients. In the computation of sensitivity, the *Allocation Optimization* module needs to compute the partial derivative $\frac{\partial f_i}{\partial \overrightarrow{u_i}}$ and $\frac{\partial f_i}{\partial \rho_i}$. Thanks to the structured neural network utilized within the latency profiling module for approximating f_i, these partial derivatives can be systematically obtained through gradient operations from the network. Given the input $\overrightarrow{x_i} = (\overrightarrow{u_i}, \rho_i, \overrightarrow{I_i})$ to the neural network, the intermediate output at the l-th layer $\overrightarrow{y_l}$ can be expressed as:

$$\begin{aligned}\overrightarrow{y_l} &= \boldsymbol{W_l}^T \cdot h(\overrightarrow{y_{l-1}}) + \overrightarrow{b_{l-1}}, \\ \text{and, } \overrightarrow{y_1} &= \boldsymbol{W_1}^T \cdot h(\overrightarrow{x_i}) + \overrightarrow{b_1}.\end{aligned} \tag{4}$$

Here, h denotes the activation function, $\{\boldsymbol{W_1}, \boldsymbol{W_2}, ..., \boldsymbol{W_k}\}$ and $\{\overrightarrow{b_1}, \overrightarrow{b_2}, ..., \overrightarrow{b_k}\}$ are the trained weights and biases across the layers of the model, where k represents the total number of layers in the neural network. Subsequently, the gradient of the last-layer output can be calculated as:

$$\mathrm{d}y_k = \boldsymbol{W_k}^T \cdot \prod_{l=1}^{k-1} \Big[\frac{\mathrm{d}h}{\mathrm{d}\overrightarrow{y_{k-l}}}\Big]_{\mathcal{M}} \cdot \boldsymbol{W_{k-l}}^T \cdot \mathrm{d}\overrightarrow{x_i}, \tag{5}$$

where $[\,\cdot\,]_{\mathcal{M}}$ represents the diagonal matrix corresponding to the vector. With Eq. (5), the partial derivative of f_i with respect to each input dimension can be directly derived from the gradient accordingly.

As containers from a single microservice may be distributed across multiple machines, diverse resource contention scenarios can arise. This diversity underscores the need to assess latency sensitivity for each container individually when making both vertical and horizontal scaling decisions. In such instances, the *Allocation Optimization* module identifies the utmost sensitivity as indicative of the entire microservice, thereby encapsulating the most critical interference scenario and potential performance bottlenecks.

C. Obtaining Feedback from Predicting E2E Latency

The *Allocation Optimization* module may conduct multiple iterations of resource scaling to fulfill the SLA concerning the tail E2E latency, particularly in scenarios characterized by significant resource contention or high loads. However, implementing each scaling decision on a physical cluster takes time to verify the resulting E2E latency, potentially resulting in multiple SLA violations throughout the scaling process. To address this challenge, the *Allocation Optimization* module acquires simulated feedback from the *E2E Latency Predictor* module to inform the scaling process.

Feature Selection. The Grad system estimates the tail E2E latency of an entire application by leveraging the latency of each individual microservice within all execution paths. However, when a microservice has multiple containers deployed across the cluster, each container may exhibit different individual tail latencies. To account for this variability and maintain scalability for the prediction model, the predictor utilizes the mean and maximum tail latency among all containers of microservice i as input features.

$$\overrightarrow{L_i} = \Big(\text{mean}\big\{l_i^1, l_i^2, \ldots, l_i^p\big\}, \max\big\{l_i^1, l_i^2, \ldots, l_i^p\big\}\Big). \tag{6}$$

Here, p denotes the number of containers for microservice i.

Self-attention Learning. The intricate interactions among microservices entail both concurrent and sequential executions, as well as mutual influences between upstream and downstream microservices. To address this complexity, our predictor incorporates a self-attention module [43] atop a neural network, as depicted in ❺ of Fig. 5. This attention mechanism enables the model to dynamically assign varying degrees of importance to the latency of each individual microservice in relation to others within the execution path. By computing a weighted sum of these representations, the self-attention mechanism allows each microservice to attend to others with distinct levels of significance. The attention matrix $\boldsymbol{A}$ is computed as follows:

$$\begin{gathered}\boldsymbol{A} = \text{Softmax}\Big(\frac{\boldsymbol{Q}\cdot\boldsymbol{K}^T}{\sqrt{d_k}}\Big)\boldsymbol{V},\\ \boldsymbol{Q}=\boldsymbol{L}\cdot\boldsymbol{W_q},\quad \boldsymbol{K}=\boldsymbol{L}\cdot\boldsymbol{W_k},\quad \boldsymbol{V}=\boldsymbol{L}\cdot\boldsymbol{W_v},\\ \boldsymbol{L}=\Big[\overrightarrow{L_1};\overrightarrow{L_2};\ldots;\overrightarrow{L_n}\Big].\end{gathered}\tag{7}$$

In this context, $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ denote the query, key, and value matrices, respectively. $\boldsymbol{L}$ represents the input matrix, synthesized by concatenating the input vectors of all n individual microservices. The dimension d_k, which equals 2 in our scenario, signifies the feature dimension. $\boldsymbol{W_Q}$, $\boldsymbol{W_K}$, and $\boldsymbol{W_V}$ are trainable weight matrices utilized to project the input matrix $\boldsymbol{L}$ into the query, key, and value spaces. The attention matrix $\boldsymbol{A}$ is linked to a fully-connected neural network, which generates the ultimate prediction for the tail E2E latency. This context-aware approach enhances the model's ability to depict the dynamics inherent in the collective performance of microservices, thereby offering a more precise and comprehensive learning of tail E2E latency.

V. Optimal Scaling with Rapid Response

In this section, we delve into the intricacies of the dynamic scaling mechanism implemented in *Allocation Optimization* module. The core principle guiding the scaling process is to adopt a fine-grained approach, which integrates both vertical and horizontal scaling during resource trading to effectively approximate an optimal scaling strategy. Operating at a finer scaling granularity enables Grad to dynamically select the most suitable microservices for scaling, thereby minimizing E2E latency to the greatest extent possible. This approach also facilitates accurate quantification of the dynamic sensitivity of each microservice to resource scaling.

The *Allocation Optimization* module of Grad acts as an iterative decision-maker, autonomously selecting microservices and making scaling decisions. It employs a combination of offline inference and feedback-driven online scheduling to promptly mitigate SLA violations. The operational flow of the scaling process within the Grad system can be divided into two distinct phases: *Prepare* and *Commit*, with the actual deployment occurring solely in the *Commit* phase.

In the *Prepare* phase, the *Monitor* module gathers environmental data from the cluster and the application's deployment configuration, including the resource usage of each microservice denoted as $\overrightarrow{u_i}$, resource contention represented by $\overrightarrow{I_i}$, and

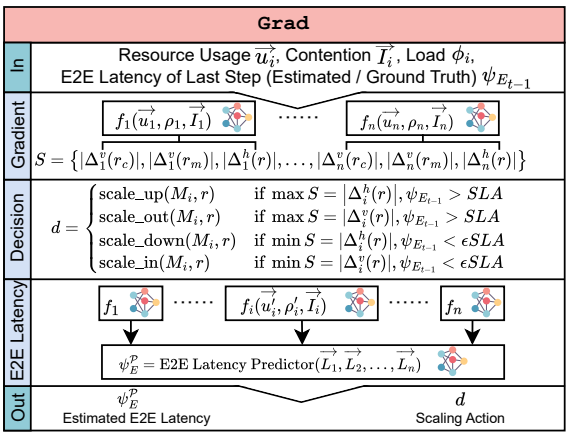
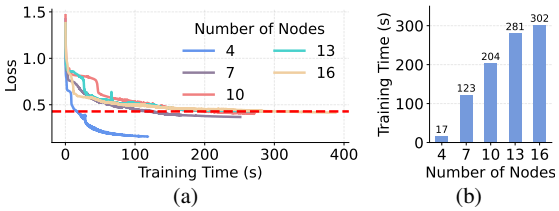


Fig. 6: Detailed computation within the `Grad` interface.

the microservice's load Φ_i. Utilizing this data, the optimization module iteratively selects an optimal set of microservices for scaling and updates the deployment configurations accordingly. Simultaneously, it incorporates feedback from the *E2E latency Predictor* module to estimate a new tail E2E latency $\psi_E^{\mathcal{P}}$ based on the updated deployment configurations. These essential steps are facilitated through the `Grad` interface, which acts as the core component of the optimization module, as depicted in Fig. 6. The main functionality of this interface is to assess the latency sensitivity of each microservice when scaling a fixed amount of resources (e.g., r = 0.2 CPU core or 100MB memory) both horizontally and vertically. This evaluation leverages the outcomes presented in Eqs. (2) and (3) to pinpoint the microservice with the highest sensitivity and determine the associated scaling methodology. Upon updating the resource configuration of a microservice, the interface promptly adjusts its sensitivity, thereby directly influencing decisions in subsequent iterations. This effectively addresses the causal relationship between consecutive selections.

Once the estimated tail E2E latency $\psi_E^{\mathcal{P}}$, from the *Prepare* phase successfully meets the SLA requirement, the operational flow transitions to the *Commit* phase. In this phase, the *Deployment* module applies the resource configurations for each scaled microservice based on the decisions made by the `Grad` interface. Following this, the *Tracing Coordinator* module gathers the actual tail E2E latency, denoted as $\psi_E^{\mathcal{T}}$, and verifies if it aligns with the SLA requirement. If the requirement is met, the entire scaling process is completed. Otherwise, the process continues, returning to the *Prepare* phase again, with the only difference being the utilization of the actual E2E latency as the new feedback. Nevertheless, experimental evaluations indicate that such reversions are rare, transpiring less than 5% of the time.

In scenarios where the microservice load decreases or resource contention diminishes, the *Allocation Optimization* module is capable of initiating scaling down or scaling in operations to conserve resources. Specifically, if the resulting tail E2E latency falls below ϵ (default 0.9) of the SLA, the `Grad` interface selects a set of microservices with minimal sensitivity to resource scaling and adjusts their resource con-



(a) (b)

Fig. 7: Training overhead as microservice scales.

figuration accordingly, as illustrated in Fig. 6. Maintaining a slack of 0.1 aims to prevent SLA violations due to occasional performance changes or mis-prediction of tail E2E latency.

VI. Implementation and Evaluation

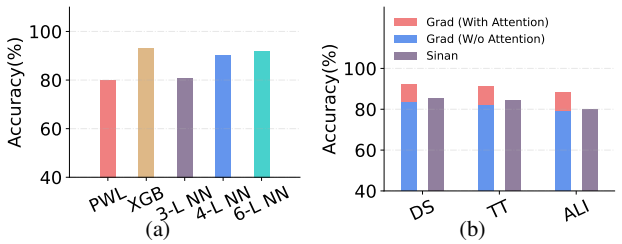
We develop a prototype of Grad atop Kubernetes, integrated with Jaeger for tracing latency samples and Prometheus for gathering low-level resource usage metrics and analyzing resource contention. The *Allocation Optimization* module is developed as a Kubernetes plugin, harnessing the parameters of the hidden layers within the trained network of the *Microservice Latency Profiling* module to calculate sensitivity, with computations executed in milliseconds.

A. Experiment Setup

Microservice Benchmarks: We evaluated Grad using two open-source microservice benchmarks: **DeathStarBench** [15] and **Train Ticket** [53]. DeathStarBench encompasses three diverse applications: Social Network, Hotel Reservation, and Media, containing 36, 15, and 38 unique microservices, respectively. Meanwhile, Train Ticket consists of three distinct applications: ticket booking, ticket querying, and food ordering, with an average of over 20 microservices invoked per application. We also utilized traces from the widely-used online shopping application, **Taobao**, provided by Alibaba [27]. The Taobao application comprises more than 2500 microservices.

Cluster Setup: We deployed Grad in a private cluster comprising 10 two-socket physical hosts. Each host is equipped with 52 CPU cores and 128 GB of RAM. Initially, each microservice container is configured with 0.2 CPU core and 100 MB of memory before scaling.

Anomaly Injection: Grad employs an Anomaly Injector to simulate various interferences that occur in real-world scenarios, including CPU, memory capacity, memory bandwidth, and network bandwidth on the host where the microservices are located. For CPU, memory capacity, and memory bandwidth interferences, Grad utilizes the Ibench tool [12] to generate anomaly containers on the injection target nodes. The type of anomaly can be controlled by the containers' consumption of the corresponding resources, while the intensity of the interference can be adjusted by the number of containers deployed. The duration of the interference can be managed by controlling the lifespan of these containers. Regarding network bandwidth anomalies, Grad employs Iperf3 [3] to enable injection target nodes to send requests to each other, thus inducing anomalies. The intensity and duration of network anomalies are also configurable. This Injector is specifically designed for the profiling phase and operates for a short



(a) (b)

Fig. 8: Accuracy of latency profiling. (a) Profiling microservice latency using piece-wise linear regression (PWL), XGBoost (XGB), and neural network (NN) with varying numbers of layers. (b) Tail E2E latency prediction on DeathStarBench (DS), Train Ticket (TT), and Alibaba traces (ALI).

duration. It helps expedites the training process of machine learning models since real-world workloads may not encounter all contention situations within a limited training period.

Offline Data Collection: For the profiling of individual microservice latency in benchmarks, we configured microservice containers with ten distinct settings for CPU (ranging from 0.1 to 1 core) and memory (from 100 MB to 1000 MB), resulting in resource utilization spanning from 10% to 80%. Meanwhile, the load varied from 20 to 1000 requests per second and each configuration was subjected to 10 sets of valid loads. Moreover, we define four distinct intensity degrees of interference for each type, where interference on CPU, memory bandwidth, and network bandwidth was normalized to a range of 30% to 80%, while memory capacity interference ranged from 30% to 60%. Specifically, to mitigate the impact of potential load imbalances across containers within the same microservice, Grad collects the 95th percentile latency for each microservice under different scenarios. This is done by first calculating the 95th percentile latency for each pod replica, and then averaging the 95th percentile latencies across all replicas within each microservice. Consequently, we amassed a dataset comprising 600,000 data samples for offline profiling, which took two hours to gather. This collection time is significantly shorter compared to existing systems [21], [35]. In Alibaba clusters, collecting sufficient data for analysis is facilitated by the availability of application log files where microservices demonstrate distinct periodic characteristics. To ensure high profiling accuracy, we require one day's worth of data from weekdays, one day's worth of data from weekends.

Load Generation: We observed that the maximum throughput of applications in our cluster is 150 (1200) requests/sec under TrainTicket (DeathStarBench). We conducted evaluations using static and dynamic loads. Regarding the static load, we generated a diverse range of requests, spanning from as low as 10 to the threshold of 150 (1200) requests per second for each application. Dynamic loads are derived from Alibaba clusters [27]. We adopted the P95 E2E latency as the target metric, aiming to keep it below SLA. And the range of SLA is set from 150 ms (low) to 500 ms (high) in the experiments.

Baseline Schemes: We compared the performance of Grad against the state-of-the art microservice scaling frameworks, including Erms [28], Sinan [49], and Firm [35].

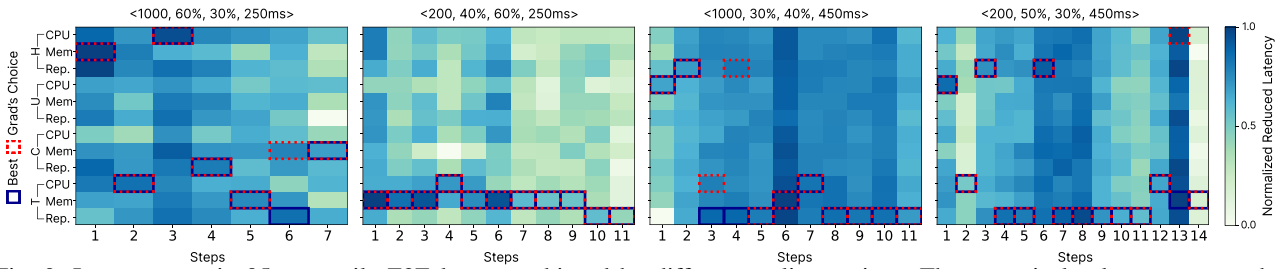


Fig. 9: Improvement in 95-percentile E2E latency achieved by different scaling options. The numerical values represent the latency improvement, normalized using the max-min method. Darker colors indicate higher improvement. H, U, C, and T respectively represent `Home-timeline`, `User-timeline`, `Compose-post`, and `Text` microservices. "CPU" and "Mem" denote the scaling up of CPU and memory, while "Rep." represents the scaling out of containers.

- **Erms:** The system profiles microservice latency using a piece-wise linear function and computes the latency target for each microservice to coordinate global horizontal scaling.
- **Sinan:** The system estimates the E2E latency using a monolithic model to facilitate resource scaling. However, it faces limitations in incorporating diverse interference conditions.
- **Firm:** The system identifies a critical microservice along each critical path that leads to SLA violations concerning E2E latency. It then utilizes reinforcement learning to optimize resource scaling for critical microservices.

B. Latency Profiling under Grad

To evaluate the accuracy of Grad's profiling modules, we conducted deployments of DeathStarBench and TrainTickets on our cluster, capturing traces for each microservice. Furthermore, we acquired traces of comparable scale from the Alibaba cluster to perform assessments in a production environment.

Training overhead. For testing purposes, we randomly sampled 15% of the dataset, reserving the remainder for model training. Larger-scale microservice applications typically require more resources to maintain robust service quality, leading to a greater number of container replicas and the need for more physical machines. This, in turn, potentially increases the amount of training data needed for model convergence. To understand how Grad's training overhead varies with increasing cluster sizes, we conducted training on microservice applications of different scales using an NVIDIA 3090 GPU, recording the convergence time of the performance model for a single microservice. As shown in Fig. 7(a), smaller-scale applications result in shorter convergence times. Fig. 7(b) shows the time required for the model to reach 90% accuracy (MAPE). Initially, training time increases with application scale, but as the scale continues to grow, the increase in fully-trained time slows down significantly, growing at a rate slower than linear. Specifically, our latency profiling model, featuring a simple network structure, can be fully trained within three minutes based on Alibaba's complex applications. Conversely, the data features for the E2E latency prediction model are less complex, requiring a training set size approximately ten times smaller than the former. It can be efficiently trained with a minute. Overall, frequent updates to microservices (usually twice a week) do not impede the retraining process.

Microservice latency estimation. To assess Grad's individual microservice latency prediction module, we compared it with two commonly-used approaches: explicit mathematical modeling including Erms' piece-wise linear regression, and more sophisticated ML algorithms like XGBoost. Additionally, we explored how prediction accuracy (using Mean Absolute Percentage Error) was influenced by adjusting the complexity of Grad's model architecture, varying the number of layers. As shown in Fig. 8(a), when employing a 4-layer neural network, Grad achieved an average accuracy of up to 90% across all benchmarks and Alibaba traces, significantly surpassing the piecewise linear method. While XGBoost achieved an accuracy of 93%, its intricate structure posed challenges in calculating latency gradients concerning different input dimensions, limiting its ability to quantify microservice sensitivity effectively. Furthermore, the accuracy of Grad's prediction network improved with additional layers, though exhibiting diminishing marginal returns. For instance, a 4-layer network demonstrated a 10% enhancement over the 3-layer counterpart, whereas a 6-layer network yielded merely a 2% improvement over the 4-layer configuration, all the while significantly escalating training and inference overhead. Hence, we settled on a 4-layer configuration for the Grad implementation.

E2E latency prediction. As depicted in Fig. 8(b), Grad achieves a prediction accuracy of up to 88% for E2E latency, outperforming Sinan by an average margin of 8.7%. This underscores its effectiveness in providing precise simulated feedback to the *Allocation Optimization* module. Furthermore, we conducted ablation experiments on the self-attention mechanism, unveiling an 11% enhancement in accuracy directly attributed to this mechanism. This notable accuracy implies that Grad's resource scaling scheme, generated solely through offline optimization algorithms during the *Prepare* phase, holds a high likelihood of meeting SLA during actual runtime.

C. Effectiveness of Grad's Scaling Optimization Mechanism

We conducted an experiment using the `compose-post` service included in the Social Network application to examine the effectiveness of Grad's scaling approach. Each microservice was initialized with default configurations (0.2 CPU core, 100MB memory, 1 container per microservice). We introduced varying loads and interference to intentionally exceed the predefined SLA for E2E latency. Subsequently, by invoking the

TABLE I: Resource usage (normalized to resource unit) under different settings among various benchmarks and Alibaba traces

Benchmarks	DeathStarBench				Train Ticket				Alibaba			
Settings \ Methods	**Grad**	Erms	Sinan	Firm	**Grad**	Erms	Sinan	Firm	**Grad**	Erms	Sinan	Firm
Low Load	8.30	9.93	11.34	14.49	17.64	28.22	29.99	44.10	14.49	21.73	34.78	35.38
High Load	21.27	27.93	37.24	53.53	69.82	111.72	141.63	157.58	65.17	104.27	169.44	143.37
Low SLA	16.93	20.31	30.47	38.93	47.39	65.82	90.04	127.95	59.24	94.78	159.94	136.25
High SLA	13.62	16.35	22.89	25.98	23.44	30.66	40.92	46.05	28.61	40.06	71.53	68.67
Low Interference	14.75	16.23	25.07	28.02	32.45	51.92	65.15	87.62	32.45	48.68	87.62	68.15
High Interference	15.87	22.21	25.39	34.91	34.91	58.87	70.83	97.27	39.67	63.47	95.20	83.30

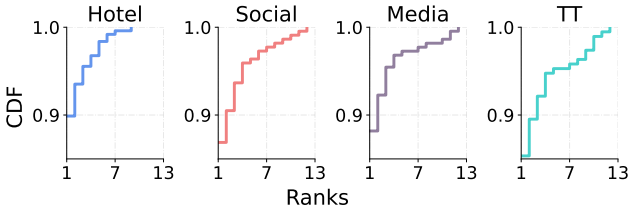


Fig. 10: Distribution of the ranking of Grad's scaling selection.

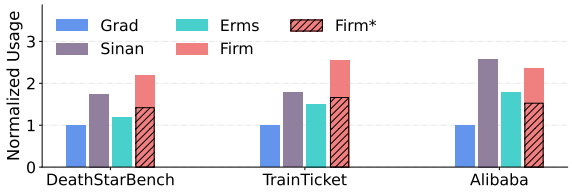


Fig. 12: Average resource usage under all benchmarks.

`Grad` interface, we aimed to make optimal scaling decisions at each iteration, ultimately achieving E2E latency levels that met the SLA. Moreover, during each iteration step, we exhaustively explored all potential scaling options, incrementing resource allocations by uniform units (0.2 CPU core or 100MB memory) or adding 1 container for each selected microservice.

The heatmaps depicted in Fig. 9 showcase the enhancements in E2E latency achieved by scaling up CPU or memory, or scaling out by increasing the number of containers for each microservice. The blocks outlined with red dashed lines represent Grad's selections at each iteration step, while those outlined with dark blue borders indicate the optimal scaling strategy. Notably, Fig. 9 presents data for only four microservices, as over 95% of the most effective scaling choices stem from these microservices. It is evident that most of Grad's selections align with the optimal choices, accounting for a percentage of 41 out of 44. Even in cases of discrepancies, Grad consistently prioritizes strategies that rank among the top five decisions.

Furthermore, we replicated the same experiment across all applications within DeathstarBench and TrainTicket. Under each application, we conducted tests across 81 diverse scenarios, scrutinizing Grad's rankings in terms of tail E2E latency improvement across all possible scaling decisions. As illustrated in Fig. 10, spanning all applications, Grad consistently identifies the optimal scaling decision in over 87.6% of cases on average, with a low probability of selecting a decision outside the top five—not surpassing 4%.

Allocating resources to the microservice with the highest latency is an intuitive scaling approach but is not always optimal. To verify this, we replaced our strategy in the iterations with selecting the microservice with the highest latency and conducted the same experiment mentioned above. As shown in Fig. 11, this approach led to better performance

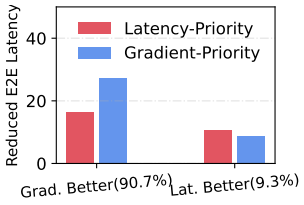


Fig. 11: Comparison between different selection criteria.

improvements in only 9.3% of cases, with an average increase of just 19.8% beyond Grad. In contrast, Grad's choices achieved over a 32.5% latency reduction in other scenarios.

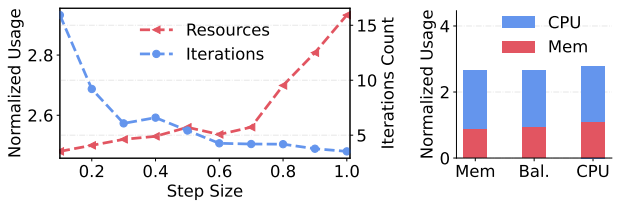
D. End-to-End Performance

1) Static loads: In this part, we present the overall resource usage and tail E2E latency under varied static loads, different degrees of interference, and diverse SLA specifications. Each benchmark underwent a 30-minute execution period.

Resource saving. We evaluated resource usage by analyzing the allocation of both CPU and memory resources, where in our cluster, 0.1 CPU core equates to 50MB memory. Fig. 12 depicts the average resource utilization across these diverse benchmarks. The results indicate that Grad consistently outperforms baselines. On DeathStarBench, Grad achieves substantial resource savings compared to Sinan, Erms, and Firm, with reductions of 42.7%, 16.1%, and 54.2%, respectively. Similarly, on TrainTicket, Grad achieves even higher resource savings, reaching 44.1%, 33.6%, and 60.9%. Notably, when evaluated using Alibaba traces, Grad achieves an impressive 54.2% reduction in resource usage on average, surpassing the improvements observed in other benchmarks.

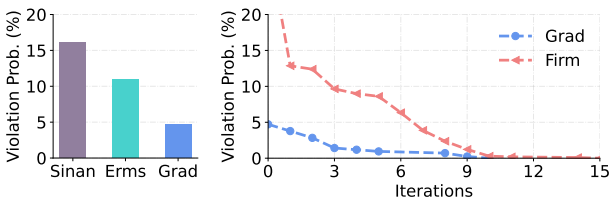
We also conducted an in-depth analysis of resource utilization across diverse settings. As depicted in Tab. I, Grad achieves remarkable savings of up to an average of 43.6% under high-load conditions, 56% higher than the reduction achieved during low-load scenarios. This trend persists across varying SLA demands; under stricter SLAs, Grad conserves an additional 15% of resources compared to situations with relaxed SLAs. When assessing different interference levels, significant variations in resource usage are observed in Alibaba traces. Accordingly, Grad achieves an average reduction of 53.2% in resource consumption in Alibaba traces, representing 20% higher savings compared to other benchmarks.

Benefit of harnessing resource fungibility. Existing techniques primarily focus on scaling critical microservices until the SLA is met. These approaches essentially narrow the search space, however, potentially resulting in sub-optimal re-



(a) (b)

Fig. 13: Tradeoff regarding the resource unit size.



(a) *Prepare* phase (b) *Commit* phase

Fig. 14: SLA violation probability.

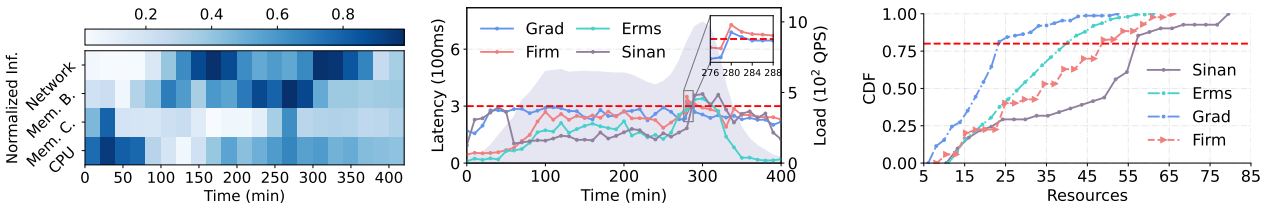
source allocation. Grad, on the other hand, considers resource fungibility between critical and non-critical microservices. To quantify this benefit more accurately, we conducted another experiment based on a slight modification of Firm's approach. Before scaling, we identified the microservice with the highest gradient from Grad. Subsequently, we employed Firm's reinforcement learning module to make scaling decisions on this non-critical microservice. We tested this across all benchmarks and Alibaba traces, with results depicted in Fig. 12. The modified Firm (Firm*) achieved a 35% resource usage savings, further demonstrating that scaling critical microservices may not be a resource-efficient approach. Despite this improvement, Firm* still used 53.2% more resources compared to Grad. This is because reinforcement learning is not fine-grained enough as it directly outputs scaling decisions that satisfy SLA. During this process, the microservice most suitable for scaling can change dynamically, as evidenced in § II-C.

Scaling overhead. Grad employs its E2E prediction module to assess whether the existing resource allocation meets SLA. In the case of DeathStarBench and Traintickets, Grad can formulate scaling plans within 5 to 30 iterations during the *Prepare* phase. For each iteration, the primary cost arises from the gradient-based ranking of microservices, which maintains a log-linear time complexity relative to the number of microservices within an application and spans no more than 10 milliseconds on a single AMD EPYC 7452 CPU core. This results in an average total response time of 130 milliseconds for all benchmarks and 300 milliseconds for Alibaba's applications. This response is significantly faster than Firm, by an order of magnitude. Given that microservice startup typically takes tens of seconds, our scheduling overhead is negligible.

The scaling process is also influenced by the iteration step size and resource trade-offs where 0.1 CPU is equivalent to 50MB memory in our setup, considering the configuration of our servers in the cluster with 64 cores and 128GB memory. To study this influence, we tested our iterative algorithm using various configurations. As shown in Fig. 13(a), the number of iterations decreases as the step size increases, while resource usage increases. This is because aggressive allocation strategies can lead to missing out on better scaling decisions. Additionally, Fig. 13(b) shows that, increasing the weight of CPU (the third bar) by 5× results in a higher proportion of memory allocation by 8%. This indicates that while meeting SLA, Grad tends to allocate more "cheap" resources.

Mitigation of SLA violation. Grad executes deployment using the allocation produced in the *Prepare* phase and gathers actual E2E latency feedback to determine the need for further adjustments during the *Commit* phase. Typically, such adjustments are unnecessary. As illustrated in Fig. 14(a), Grad maintains an average SLA violation probability below 4.7% across all benchmarks, notably lower than the 10.9% and 6.1% observed with Sinan and Erms. In instances of violations, Grad ensures that over 70% can be rectified within three additional iterations to adhere to the SLA. Furthermore, the worst-case scenario necessitates no more than 10 iterations, marking a 46% reduction compared to Firm, as depicted in Fig. 14(b).

2) Dynamic loads: To assess Grad's efficiency in resource saving within intricate production environments, we conducted a 400-minute experiment utilizing dynamic loads and varied interferences, and setting a SLA target of 300ms. The fluctuating trends of both interference and loads were sampled from the Taobao application in Alibaba traces. This sampling was conducted over a 400-minute period from a typical workday, capturing the average interference conditions across 6000 physical hosts. The experiment centered on the Hotel-reservation application. Additionally, to gauge Grad's performance under extreme pressure, we induced a sudden surge in both load and interference at the 280th minute, which was followed by a significant fluctuation persisting until the 300th minute. Fig. 15(a) displays the fluctuation of four types of interferences over time, while the shaded area in Fig. 15(b) illustrates temporal load variations. Concurrently, Fig. 15(b) also depicts the tail latency of requests over the 400-minute duration. All evaluated schemes demonstrated responsiveness to load and interference fluctuations and were able to control E2E latency to meet SLA requirements, before extreme scenarios of rapid load and interference escalation. As depicted in the magnified region in Fig. 15(b), both Grad and Firm rapidly responded to the spike, leveraging feedback information. Grad successfully maintained tail E2E latency within SLA in 4 minutes. Moreover, it only marginally violates the SLA by less than 4% during this timeframe. However, Firm struggled to manage these scenarios characterized by frequent load and interference changes, leading to continuous SLA violations (violate the SLA by 10%), lasting for 20 minutes. This is due to the nature of Firm, which necessitates the collection of actual E2E latency data, and the detection of SLA violations can only occur once the delayed requests are executed. Even if resources are promptly increased upon detecting a violation, there is still a delay in emptying the existing queues. Furthermore, during this draining process, the environment may undergo further changes. In contrast, Grad has the ability to predict E2E latency in the current state without relying on real-time data collection, allowing for



(a) (b) (c)

Fig. 15: Experiment with dynamic loads and varying interferences. (a) Interference fluctuations over time, with darker colors indicating higher levels of interference. Network, Mem. B., Mem. C., and CPU represent the interference of network traffic, memory bandwidth, memory capacity, and CPU usage. (b) P95 E2E latency fluctuates over time. (c) CDFs of resource usage.

more timely responses and reducing the frequency of SLA violations. Sinan and Erms suffered extended SLA violations post-peak load (35 and 16 minutes, respectively). This was attributed to Sinan's poor adaptability to high interference and Erms' oversimplified interference model.

Grad intelligently manages resources to closely align tail E2E latency with the SLA. This fine-grained approach yields significant cost savings, as illustrated in Fig. 15(c), which highlights Grad's strong capacity to trim the tail end of resource usage distribution. Notably, within 80% of the time, Grad saves the overall resource allocation by up to 42.5%, 53.1%, and 59.6% compared to Erms, Firm, and Sinan.

To highlight the role of resource fungibility, we focus on one state transition between 260 and 280 minutes, comparing Firm's and Grad's decisions. During that period, Firm selected `Reservation` as critical microservice and allocated 6 units of memory to maintain QoS. Grad, on the other hand, allocated 2 units of memory to `Reservation` and 1 unit of CPU each to `Frontend` and `Search`. Consequently, Grad's decisions reduce E2E latency by 12% per iteration on average compared to Firm as a result of its consideration of resource fungibility.

VII. Related Work

Microservice Performance Modelling. Graf [34] adopts monolithic models, which integrate all microservices into a unified E2E learning framework. However, these models encounter difficulties in accurately capturing diverse interference conditions within a shared environment. GrandSLAm [19] calculates the average execution time of individual microservices to establish performance profiles. Meanwhile, ORION [30] employs ML to profile latency across heterogeneous resources, thereby requiring assessment across numerous configurations. Moreover, Erms [28] applies a straightforward piece-wise linear function to profile individual microservice latency, which cannot adequately address the multiple resource usage factors that influence microservice performance.

Root Cause Diagnosis. Existing studies leverage heuristic algorithms or ML techniques to detect performance anomalies and analyze their root causes [14], [16], [18], [24], [25], [35], [37], [41], [45], [46]. In particular, Firm [35] utilizes SVM to identify critical microservices with significant impacts on overall service performance. Meanwhile, Sage [14] constructs a graphical model to identify the root causes of unpredictable microservice performance and dynamically adjusts resources. Seer [16] employs deep learning to recognize spatial and temporal patterns associated with SLA violations. However, these endeavors primarily concentrate on adjusting anomalous microservices, potentially overlooking the resource fungibility between critical and non-critical microservices.

Microservice Autoscaling. Numerous microservice autoscaling approaches have been proposed in the literature [7], [8], [10], [14], [17], [19], [28], [31]–[33], [35], [36], [47]–[49], [51]. GrandSLAm [19] and Rhythm [51] quantify the contribution of each microservice to E2E latency. Ursa [50] uses an analytical model to decompose the E2E SLA into per-microservice SLAs and maps them to resource allocations at the individual microservice level, but it overlooks the opportunities offered by the fungibility of resources across microservices. Autothrottle [44] decouples application-level SLO feedback and service-level resource control by employing a novel metric—CPU throttles—to drive per-service resource management with local performance targets. Nonetheless, it falls short in that it can only scale CPU resources.

VIII. Conclusion

We have designed an intelligent, interference-aware resource management system tailored for scaling microservices within shared clusters. This innovative system underscores the benefits of effectively leveraging resource fungibility between critical and non-critical microservices. Furthermore, it adeptly tackles the challenges stemming from the dynamic nature of resource fungibility during scaling operations by employing a modular learning approach to profile microservice latency. Additionally, it adopts a fine-grained optimization scheme to select the optimal set of microservices for resource trading. These design features empower the system to promptly address instances of SLA violation or resource over-provisioning.

IX. Acknowledgment

We sincerely thank the anonymous HPCA'25 reviewers for their valuable suggestions that improved the paper. This work is supported by the Science and Technology Development Fund of Macau (0024/2022/A1, 0071/2023/ITP2, 0081/2022/A2, and 0123/2022/AFJ), as well as the Multi-Year Research Grant of University of Macau (MYRG-GRG2023-00019-FST-UMDF, MYRG-GRG2024-00255-FST-UMDF)

References

[1] "Cncf:," https://www.cncf.io/.

[2] "Alibaba microservices cluster traces." https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021, 2021.

[3] "Iperf," https://iperf.fr/, 2022.

[4] "Jaeger," https://jaegertracing.io/, 2022.

[5] "Prometheus," https://prometheus.io/, 2022.

[6] N. Bashir, N. Deng, K. Rzadca, D. Irwin, S. Kodak, and R. Jnagal, "Take it to the limit: Peak prediction-driven resource overcommitment in datacenters," in *Proceedings of EuroSys*, 2021.

[7] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *Proceedings of SoCC*, 2021.

[8] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of ASPLOS*, 2019.

[9] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of SIGKDD*, 2016.

[10] K.-H. Chow, U. Deshpande, S. Seshadri, and L. Liu, "Deeprest: deep resource estimation for interactive microservices," in *Proceedings of EuroSys*, 2022.

[11] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of SOSP*, 2017.

[12] C. Delimitrou and C. Kozyrakis, "Ibench: Quantifying interference for datacenter applications," in *Proceedings of IISWC*, 2013.

[13] G. K. Engine, https://cloud.google.com/kubernetes-engine, 2022.

[14] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ml-driven performance debugging in microservices," in *Proceedings of ASPLOS*, 2021.

[15] Y. Gan, Y. Zhang *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of ASPLOS*, 2019.

[16] ——, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of ASPLOS*, 2019.

[17] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *Proceedings of ICDCS*, 2019.

[18] S. Jha, S. Cui, S. S. Banerjee, T. Xu, J. Enos, M. Showerman, Z. T. Kalbarczyk, and R. K. Iyer, "Live forensics for hpc systems: A case study on distributed storage systems," in *Proceedings of SC*, 2020.

[19] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of EuroSys*, 2019.

[20] Kubernetes, https://kubernetes.io., 2022.

[21] C.-J. M. Liang, Z. Fang, Y. Xie, F. Yang, Z. L. Li, L. L. Zhang, M. Yang, and L. Zhou, "On modular learning of distributed systems for predicting end-to-end latency," in *Proceedings of NSDI*, 2023.

[22] M. Liang, Y. Gan, Y. Li, C. Torres, A. Dhanotia, M. Ketkar, and C. Delimitrou, "Ditto: End-to-end application cloning for networked cloud services," in *Proceedings of ASPLOS*, 2023.

[23] C. Lin, S. Luo, and H. Xu, "Exploring imbalances among microservice containers in large cloud platforms," in *Proceedings of ISPA*, 2023.

[24] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Proceedings of ICSOC*, 2018.

[25] H. Liu, J. Zhang, H. Shan, M. Li, Y. Chen, X. He, and X. Li, "Jcallgraph: tracing microservices in very large scale container cloud platforms," in *Proceedings of CLOUD*, 2019.

[26] C. Lu, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "Understanding and optimizing workloads for unified resource management in large cloud platforms," in *Proceedings of EuroSys*, 2023.

[27] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of SoCC*, 2021.

[28] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "Erms: Efficient resource management for shared microservices with sla guarantees," in *Proceedings of ASPLOS*, 2023.

[29] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "The power of prediction: Microservice auto scaling via workload learning," in *Proceedings of SoCC*, 2022.

[30] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "Orion and the three rights: Sizing, bundling, and prewarming for serverless dags," in *Proceedings of OSDI*, 2022.

[31] A. Mirhosseini, S. Elnikety, and T. F. Wenisch, "Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices," in *Proceedings of SoCC*, 2021.

[32] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch, "Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices," in *Proceedings of HPCA*, 2020.

[33] J. Ortiz, B. Lee, M. Balazinska, J. Gehrke, and J. L. Hellerstein, "Slaorchestrator: Reducing the cost of performance slas for cloud data analytics," in *Proceedings of USENIX ATC*, 2018.

[34] J. Park, B. Choi, C. Lee, and D. Han, "Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices," in *Proceedings of CoNEXT*, 2021.

[35] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: An intelligent fine-grained resource management framework for slo-oriented microservices," in *Proceedings of OSDI*, 2020.

[36] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in *Proceedings of SoCC*, 2021.

[37] S. Y. Shah, X.-H. Dang, and P. Zerfos, "Root cause detection using dynamic dependency graphs from time series data," in *Proceedings of IEEE BigData*, 2018.

[38] A. Sriraman and T. F. Wenisch, "μ suite: A benchmark suite for microservices," in *Proceedings of IISWC*, 2018.

[39] ——, "μtune: Auto-tuned threading for oldi microservices," in *Proceedings of OSDI*, 2018.

[40] C. Tang, K. Yu, K. Veeraraghavan, J. Kaldor, S. Michelson, T. Kooburat, A. Anbudurai, M. Clark, K. Gogia, L. Cheng, B. Christensen, A. Gartrell, M. Khutornenko, S. Kulkarni, M. Pawlowski, T. Pelkonen, A. Rodrigues, R. Tibrewal, V. Venkatesan, and P. Zhang, "Twine: A unified cluster management system for shared infrastructure," in *Proceedings of OSDI*, 2020.

[41] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems," in *Proceedings of Middleware*, 2017.

[42] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *Proceedings of IISWC*, 2016.

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[44] Z. Wang, P. Li, C.-J. M. Liang, F. Wu, and F. Y. Yan, "Autothrottle: A practical bi-level approach to resource management for slo-targeted microservices," in *Proceedings of NSDI*, 2024.

[45] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1646–1659, 2018.

[46] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Microrca: Root cause localization of performance issues in microservices," in *Proceedings of APNOMS*, 2020.

[47] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt, "Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows," in *Proceedings of ICDCS*, 2019.

[48] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proceedings of ICWS*, 2019.

[49] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *Proceedings of ASPLOS*, 2021.

[50] Y. Zhang, Z. Zhou, S. Elnikety, and C. Delimitrou, "Ursa: Lightweight resource management for cloud-native microservices," in *Proceedings of HPCA*, 2024.

[51] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao, "Rhythm: component-distinguishable workload deployment in datacenters," in *Proceedings of EuroSys*, 2020.

[52] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of SoCC*, 2018.

[53] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of ICSE*, 2018.