RESEARCH ARTICLE

LSRAM: A Lightweight Autoscaling and SLO Resource Allocation Framework for Microservices Based on Gradient Descent

Kan Hu[1] | Minxian Xu[1] | Kejiang Ye[1] | Chengzhong Xu[2]

[1]Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China | [2]State Key Lab of IOTSC, Department of Computer Science, University of Macau, Macau SAR, China

Correspondence: Minxian Xu (mx.xu@siat.ac.cn)

Received: 28 May 2024 | **Revised:** 29 September 2024 | **Accepted:** 15 November 2024

Funding: This work is supported by the National Natural Science Foundation of China (No. 62072451, 62102408, 92267105), Guangdong Basic and Applied Basic Research Foundation (No. 2024A1515010251, 2023B1515130002), Guangdong Special Support Plan (No. 2021TQ06X990), Shenzhen Basic Research Program under grants JCYJ20220818101610023, and JCYJ20240809180935001, Shenzhen Industrial Application Projects of undertaking the National key R & D Program of China (No. CJGJZD20210408091600002).

Keywords: gradient descent | lightweight | microservices | resource autoscaling | SLO allocation

ABSTRACT

Objective: The microservices architecture has become a dominant paradigm in cloud computing due to its advantages in development, deployment, modularity, and scalability. Ensuring Quality of Service (QoS) through efficient Service Level Objective (SLO) resource allocation is a critical challenge. Current frameworks for microservice autoscaling based on SLOs often rely on heavy and complex models that are time-consuming and resource-intensive, making them unsuitable for rapidly changing environments and highly dynamic workloads.

Methods: This study proposes LSRAM (Lightweight SLO Resource Allocation Management), a novel framework designed to overcome the limitations of existing SLO-based autoscaling methods. LSRAM operates in two stages:

1). Lightweight SLO Resource Allocation Model: Computes optimal SLO resource allocation for each microservice using a gradient descent method, ensuring rapid computation and minimal computational overhead.

2). SLO Resource Update Model: Adapts resource allocation dynamically in response to changes in the cluster environment, such as varying loads and application types, without requiring extensive retraining.

Results: LSRAM effectively addresses scenarios involving bursty traffic and fluctuating workloads. Compared to state-of-the-art SLO allocation frameworks, LSRAM achieves the following:

1). Reduces resource usage by 17%.

2). Maintains QoS guarantees for users, even under dynamic conditions.

3). Demonstrates faster adaptability to changes in the system environment due to its lightweight design.

Conclusion: LSRAM offers a scalable, efficient, and adaptive solution for SLO-based resource allocation in microservices architectures. By reducing resource usage while maintaining QoS, it provides a robust framework for managing dynamic and unpredictable workloads in cloud environments. Its lightweight design ensures practical applicability and superior performance compared to traditional, resource-intensive methods.

1 | Introduction

In contrast to the traditional monolithic architecture, where all functional components are integrated, microservices architecture decouples a complex application into multiple independent and lightweight distributed components [1]. Each component is responsible for specific functionalities, facilitating efficient management, maintenance, and updates [2]. Therefore, the use of microservices architecture in the cloud computing paradigm has become a trend, widely adopted by major enterprises such as Amazon [3], Netflix [4], Alibaba [5], [6]. Due to the lightweight and loosely coupled nature of microservices architecture, each functional component operates independently [7]. As the architecture handles increasing workloads, system administrators can identify the components bearing the load and independently scale them, rather than expand resources for the entire cloud service application.

For users of cloud service applications, their primary concern lies not in the specific operational details of backend services but in their own service experience. Users have varying tolerance levels for response time (RT) across different applications. For instance, Amazon found every 100 ms of latency would cost them 1% in sales, and Google conducted a study revealing that a delay of 0.5 s in the RT of a search page could lead to a 20% decrease in traffic [8]. Consequently, cloud service providers establish SLO metrics [9], imposing constraints on the mean or tail latency distribution of end-to-end latency within cloud services. This is achieved to ensure QoS [10] remains within acceptable ranges.

Cloud service users prioritize their QoS, leading to numerous studies focusing on the allocation of SLOs [11]. These studies define a partial SLO that represents expected targets for latency in specific parts among the application for individual microservice components to ensure the end-to-end SLO is met if all partial SLOs are satisfied [12]. Existing methodologies involve the definition of partial SLOs based on empirical knowledge, reinforcement learning techniques, or the application of deep learning algorithms to compute and design partial SLOs [13]. However, a significant limitation of these methods is that neural networks tend to become excessively large with the increase in the microservices topology. In addition, a single microservice can be shared among multiple services, each with diverse workload patterns and different SLOs, for such shared microservices, it is challenging to define their SLO resource allocation. These result in slow computation speeds and substantial consumption of system resources. Additionally, these approaches struggle to effectively respond to sudden surges in traffic.

To address the above limitations, we propose LSRAM, a Lightwight SLO Resource Allocation Management framework for highly volatile load environments based on SLO allocation. LSRAM takes the load latency characteristics of microservices and the relationship between microservices as model inputs and rapidly derives allocation scenarios through the lightweight SLO resource allocation model. During runtime, LSRAM continuously monitors changes in microservice applications and loads in the cluster, updating the SLO resources of each microservice in real-time to make them more suitable for the current cluster environment. For highly volatile loads, LSRAM implements a prediction model that can predict the next time period load in advance and respond quickly to resource scaling while further considering cascading dependencies and taking instance creation time into account in the resource scaling architecture to further ensure the QoS for users.

Compared to traditional methods of CPU and memory resource allocation, treating SLO resources as a system resource allocated to each microservice more effectively ensures the quality of user services. While an appropriate CPU utilization threshold can improve overall system resource efficiency, it cannot effectively guarantee the quality of user services. The metrics and optimization goals of these two approaches diverge from the outset: CPU resource allocation aims to maintain a high level of system resource utilization, whereas SLO resource allocation focuses on maximizing system resource savings while ensuring user service quality. Moreover, through an effective SLO resource allocation model, we can identify the optimal SLO resource allocation scheme for the current environment, ensuring that microservices minimize system costs while maintaining service quality.

In addition, we built a prototype system of LSRAM based on Kubernetes and evaluated LSRAM by deploying realistic microservice applications and conducting high fluctuation load test evaluations, and the experimental results show that LSRAM can reduce the CPU resource consumption by about 17% compared with the existing methods and has more significant performance in reducing SLO violations. Our key contributions are as follows:

- We design a lightweight, tail-latency-based SLO resource allocation model and propose appropriate computational methods for shared microservices. This model can quickly compute an SLO resource allocation scheme.
- We propose an SLO resource update model that allows the SLO resource allocation scheme to be updated in real-time based on load traffic changes in the cluster during runtime, ensuring the effectiveness of the system's runtime.
- We optimize a neural network algorithm for load prediction, enhancing the framework's ability to cope with bursty traffic. This optimization effectively guarantees end-to-end SLOs of microservices and improves user experience. Additionally, we have designed a high-volatility mode to effectively cope with high-volatility loads to ensure QoS.

The remainder of this paper is organized as follows: Section 2 discusses the related work, which focuses on the related research work on autoscaling techniques for SLO resource management in microservices architecture. Section 3 presents the general framework of LSRAM. In Section 4, the SLO resource allocation model is described in detail, and the entire LSRAM workflow is presented. Section 5 introduces the settings of the experiment and the analysis of the results. Section 6 summarizes the main findings of the paper and outlines future work plans.

2 | Related Work

Autoscaling [14] techniques for resource management in microservices architecture is a popular and attractive topic.

Due to the importance of SLO metrics in cloud services, the study of SLO-oriented and guaranteeing QoS for users plays an important role in microservice autoscaling techniques. In the following subsections, we summarize various methods for ensuring SLOs across different types of scaling approaches.

2.1 | Rule-Based Autoscaling Approaches to Satisfy Slos

Rule-based autoscaling approaches use the utilization of predefined thresholds and heuristics to initiate scaling actions [15, 16]. Most cloud platforms, such as Google Cloud and Amazon Web Services, employ this approach. A well-known method is Horizontal Pod Autoscaling (HPA), integrated with the Kubernetes platform [17], which scales resources by defining an average CPU utilization threshold for microservices. This approach is simple and straightforward; the key is to find an appropriate threshold point to balance the relationship between RT and resource consumption. Most cloud platforms configure their thresholds based on historical experience. Google's autoscaling framework, Autopilot [18], which monitors the RT of the request to access the cloud service in real-time, when the RT is much larger or less than the predefined SLO threshold, the framework will automatically change the number of instances in a timely manner to expand or decrease the number of services and meet the SLO indicators with the lowest resource consumption.

To better adapt the scheduling threshold to the cluster load environment, many studies propose a dynamic threshold approach, which scales microservices with different thresholds under varying load conditions. Liu et al. [19] propose a multi-model controller that inherits adaptive decisions from multiple models to select the best scenario; the main advantage of the empirical model is that it obtains high-quality performance predictions based on empirical data. For new application scenarios, this controller takes other models or heuristics as a starting point and continuously optimises this empirical value by continuously incorporating all performance data into the knowledge base of the empirical model.

The main goal of these SLO-oriented methods is to determine the optimal threshold value, but they face challenges in handling intricate workload dynamics, adapting to rapidly changing conditions, and optimizing resource allocation under complex scenarios.

2.2 | Learning-Based Autoscaling Approach to Satisfy Slos

There exists a large number of learning-based methods for autoscaling to satisfy SLOs. We summarise these methods into two categories based on the objects they learn from.

Historical data-driven learning-based methods: These approaches focus on analysing and modeling historical data in a microservices architecture [20, 21]. They devise a series of centralized, data-driven models to address resource scaling for microservices. Seer [22] analyzes massive amounts of tracing data collected by cloud systems to learn spatial and temporal patterns, which are then translated into QoS violations using machine learning. Sage [23] is a supervised machine learning-based system that models the dependencies between the microservices using causal Bayesian networks to identify the root cause of SLO violations. While Sage can mitigate SLO violations for microservices, it lacks the ability to adapt to changes in workloads. However, Sage can predict potential QoS violations based on the underlying data it monitors and allocate resources in advance to ensure QoS.

Environmental interaction exploration-driven learning-based methods: This category of research focuses on autonomously learning optimal scaling strategies by iteratively exploring interactions with the environment [24–26]. Coscal [27] takes advantage of data-driven decision-making by efficiently learning the scaling techniques for system resources and improving the RT of scaling techniques. However, Coscal relies heavily on Q-learning; the speed and accuracy of the results it generates decrease dramatically as the amount of data increases. AutoMan [28] introduces a resource allocation strategy employing multi-agent deep deterministic policy gradient reinforcement learning. This method aims to ensure that microservices meet end-to-end tail latency SLOs. FIRM [29] is an intelligent, fine-grained resource management framework that focuses on finding and removing contention in the microservice that is the critical cause of latency SLO violation. However, FIRM lacks the ability to identify configurations, which may result in excessive resource allocation to such microservices, leading to resource wastage. GRAF [30] proposes a graph neural network-based proactive resource allocation framework that minimizes overall CPU resources in the microservice chain while satisfying latency SLO. However, when faced with large-scale microservice architectures, GRAF can be inefficient and unable to achieve real-time allocation, while its inability to resolve situations related to microservice resource contention. In addition, its resource allocation scheme is not an optimal solution.

In summary, learning-based autoscaling approaches to ensure SLO mostly share some common drawbacks, including the inability to quickly adapt to changes in microservice application architectures and poor scalability. Many models consume a lot of resources for computation, which is not efficient for a highly fluctuating environment.

2.3 | Model-Based Autoscaling Approaches to Satisfy Slos

Some studies also employ formal mathematical models to establish relationships between workload characteristics and resource provisioning [37, 38].

Epma [32] designs two modules to detect and identify the root cause of the overload state at the application and a variety of scheduling strategies to address the identified problems. However, the resource allocation scheme given is not the optimal solution for the system, and the heuristic algorithm makes it unable to cope with unexpected loads. Hansel [33] analyzes the historical resource usage of microservice container pods and models it alongside service quality monitoring SLO information. The model is utilized for proactive scaling control and reactive scaling control decisions based on current SLOs. However, the models of Hansel need constant updates for training, which cannot quickly incorporate changes in microservice applications, and its resource allocation tends to over-scale to guarantee SLOs. Showar [34] models and analyzes microservice resource scaling in two directions: vertically using the empirical variance of historical resource usage to find the optimal size, and horizontally using cybernetic methods to find the optimal configuration. GrandSLAm [35] estimates the completion time for a single microservice component to propagate a request in a microservice application and uses this estimate to drive the scaling of system resources. However, it does not consider coordination, and its estimate may not be optimal. Luo et al. [7] introduces Erms, a resource management system designed to ensure SLOs in a shared microservice environment. Erms profiles microservice latency as a piecewise linear function of the workload, resource usage, and interference. It constructs resource scaling models to optimally define latency targets for microservices with intricate dependencies. However, Erms tends to over-provision resources for online services with highly dynamic dependency graphs and requires explicit modeling for each graph, making it challenging to achieve comprehensiveness.

Unlike the aforementioned work focusing on autoscaling resources to ensure SLO, our approach considers SLO as the allocation of resources to guarantee the SLO of different microservices. Our work is most similar to Parslo [36], a gradient descent-based approach to assign partial SLOs among nodes in a microservice graph under an end-to-end latency SLO. Parslo isolates different nodes in the Directed Acyclic Graph (DAG) from each other, enabling each microservice to scale independently through its autoscaling framework. It employs an approximate optimal partial SLO allocation scheme for minimizing overall deployment costs in general microservice graphs. However, for larger microservice call graph structures, the iteration time increases, requiring substantial resources for both offline and online SLO allocation verification. Parslo necessitates similar load-latency profiles for all microservices, yielding effective results for specific structures but limiting applicability in diverse scenarios. It is not suitable for high dynamic loads and lacks a robust strategy for sudden changes in traffic.

Table 1 shows the comparison between our work and the related work. The main differences between LSRAM and these approaches lie in three aspects. Firstly, LSRAM designs a lightweight SLO resource allocation approach that can quickly obtain the resource allocation scheme. Secondly, LSRAM quickly adapts to changes in the cluster environment and microservice applications by the update algorithm. Thirdly, LSRAM enhances adaptability to dynamic loads and traffic bursts with load prediction.

3 | System Model

In this section, we will show the design of the overall system architecture of LSRAM, as shown in the Figure 1, the key modules are as follows:

Load Monitor: The Load Monitor is designed to gather real-time load data for the current microservice deployed in the cluster (via Prometheus[1]). It generates a dependency graph for each microservice (via Jaeger[2]) and tallies the number of requests flowing to various branches of the microservice over a specified time period. During the initialization phase, the load monitor senses the topology of the entire microservice and sends it to the SLO resource allocator to assist with the initial allocation of SLO resources. During runtime, the load monitor

TABLE 1 | Comparison of related work.

Work	Load prediction	Model complexity		Updated resource allocation	SLO guarantee	SLO as Allocation Resources
		Heavy	Lightweight			
HPA [31]			✓			
FuSys [19]			✓	✓		
Autopilot [18]			✓			
Seer [22]	✓	✓				
Sage [23]	✓	✓			✓	
Coscal [27]	✓	✓				
FIRM [29]		✓		✓		
AutoMan [28]	✓	✓			✓	
GRAF [30]	✓	✓			✓	
Epma [32]			✓			
Hansel [33]			✓			
Showar [34]			✓			
GrandSLAm [35]			✓			✓
Erms [7]	✓	✓			✓	
Parslo [36]		✓		✓		✓
LSRAM (This paper)	✓		✓	✓	✓	✓

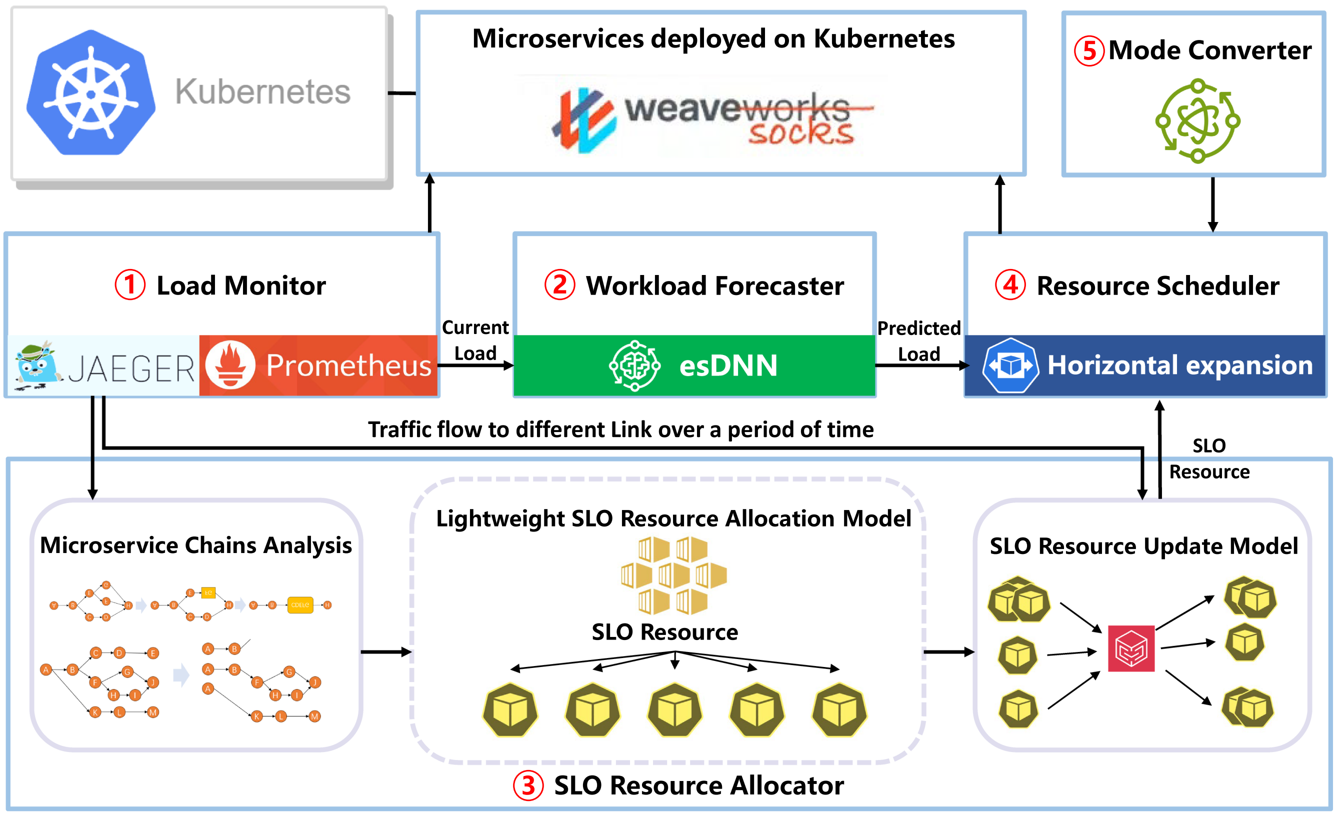


FIGURE 1 | System model of LSRAM.

continuously collects the number of requests flowing through different microservice chains and sends the traffic distribution to the SLO resource allocator regularly. It serves as the top-level awareness of the entire framework, providing inputs necessary for quickly and accurately representing the system status. Since we only invoke APIs of the relevant tools, the load monitoring tool and the tracing tool in this component can be easily replaced with other tools offering similar functionalities.

Workload Forecaster: In order to cope with the changes of load traffic in the cluster and to enhance the system's ability for handling unexpected traffic, we design a workload Forecaster for predicting the load changes and pre-allocating resources in advance to ensure the QoS. The Workload Forecaster optimizes our previous work, named esDNN [39], a deep learning network based on supervised learning, by intensive training for highly volatile loads and bursty traffic loads. Additionally, we deploy esDNN as the prediction model to forecast the load of the next time slice by processing cluster load data obtained from the Load Monitor. To be noted, this prediction model can be easily replaced with other lightweight prediction models.

SLO Resource Allocator: The lightweight and accurate allocation of SLO resources is one of the core functions of the LSRAM framework. It comprises a microservice chain analyzer, a lightweight SLO resource allocation model, and an SLO resource update model. These components efficiently compute the allocation of SLO resources based on end-to-end tail latency (eTL), leveraging load-latency-profile (LLP, which indicates the relationship between the request arrival rate and the eTL) graphs and microservice chain structure diagrams. Among these, the lightweight SLO resource allocation model based on eTL is the core model of LSRAM, which solves the shortcomings of other methods in calculating the SLO allocation scheme, which is time-consuming and consumes a lot of computational resources. After initialization, the existence of the SLO resource update model can make LSRAM quickly adapt to the changes in the cluster environment and ensure user satisfaction. During the subsequent runtime of the framework, the SLO Resource Allocator will weight each microservice according to the load traffic direction in a period of time sent by the Load Monitor and reallocate SLO resources, so as to make the whole framework system more adaptable to the current system traffic changes, ensure service quality, and save system resources at the same time.

Resource Scheduler: Resource scheduling is based on the resource computed by SLO Resource Allocator as the scaling threshold, and the QoS is guaranteed based on the number of load-scaling instances predicted by Workload Forecaster for the next time slice. When the predicted load exceeds the allocated resource threshold, the resource scheduler scales the microservice instance ahead of time based on the number of thresholds exceeded. The Resource Scheduler will allocate the scheduling resources in advance according to the predicted value, so that when the load arrives in the next time period, there are sufficient seasonal celery resources to cope with it and ensure the QoS.

Mode Converter: The Mode Converter is designed to cope with specific high-volatility load conditions. Equipped with a built-in oscillation-aware function, it monitors real-time load obtained from the Load Monitor. This function collects load values over a period of time, and when the oscillation value surpasses a predefined threshold, LSRAM identifies the cluster as being in a high-volatility state. Subsequently, the Mode Converter proactively switches the system resource scaling mode, rapidly scaling resources to meet user service demands.

LSRAM efficiently allocates suitable SLO resources to each microservice using the above mentioned components and accurately predicts upcoming load changes to proactively allocate resources during runtime. This framework resolves the issue of excessive time and computational resource consumption associated with the eTL SLO allocation method. Instead, it introduces a lightweight computational approach that effectively handles bursty traffic and significant load fluctuations. Moreover, each module is independent of the others, so besides the SLO resource allocator, all other modules can be easily replaced to ensure the flexibility of our proposed framework.

4 | Algorithms Developed in LSRAM

In this section, we introduce the overall workflow of LSRAM and focus on the algorithms deployed in LSRAM, including the lightweight SLO resource allocation model and the SLO resource update algorithm.

4.1 | Workflow of LSRAM

The overall workflow of LSRAM is illustrated in Algorithm 1. LSRAM allocates appropriate SLO resources s_i for each microservice based on the microservice structure provided by the load monitor through the SLO resource initial allocation algorithm (lines 1–3), laying the groundwork for resource autoscaling. During regular runtime (lines 6–15), the load monitor transmits the monitored load situation of the current cluster to the workload predictor to predict the load in the next time period. The resource scheduler pre-allocates resources based on the predicted load value. After some time periods or cluster environment changes, the SLO resource allocator executes the SLO resource update model to update the SLO resources for shared microservices and other microservices (lines 16–19). If the mode converter detects a high load fluctuation (lines 20–23), it adjusts the resource scheduling strategy to ensure user service satisfaction.

Complexity Analysis: Since Algorithm 1 is the scheduling algorithm for the entire system framework, its time complexity is composed of the time complexities of its various components. First, the SLO Resource Initialization Allocation has a time complexity of $O(n \log n)$ (detailed in Section 4.3). The Cluster Runtime and mode converter have a time complexity of $O(1)$, where they collect data over a period and compare it with the oscillation coefficient threshold. We utilized a queue to store the data. The update phase has a time complexity of $O(m * n)$, where m denotes the number of microservices that need to be updated. Therefore, the total time complexity of Algorithm 1 is $O(n \log n + m * n)$.

ALGORITHM 1 | LSRAM overall scheduling algorithm.

```
Input: Current load $CL$
Output: Scaling strategies to meet SLO
1  SLO resource initialization allocation:
2    Calling Algorithm 2 to initialize SLO allocation;
3    SLO resources $s_i$ allocated to each microservice;
4  Initialize parameters:
5    Current Load $CL$, future load $FL$, time index $t$, clustering
     oscillation factor $O$, predefined oscillation coefficient
     thresholds $OCT$, predefined resource reallocation time
     thresholds $M_t$, number of instances $instance_i$ of each
     microservice, the load scaling threshold $f(s_i)$ corresponds to
     the SLO resources $s_i$.
6  Cluster runtime:
7  $O = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}[CL_i - \bar{CL}]^2}}{\bar{CL}}$
8  if $O \leq OCT$ then
9      Each microservice $i$ autoscaling by $s_i$;
10     $FL \leftarrow$ Predicted by esDNN model $\leftarrow CL$
11     if $FL \geq f(s_i)$ then
12         $instance_i = \lceil \frac{FL}{f(s_i)} \rceil$
13         Scaling operation;
14 Update:
15 if $t \geq M_t$ or cluster environment changes then
16     Calling Algorithm 3 to update SLO resource;
17 Mode Converter:
18 if $O \geq OCT$ or cluster environment changes then
19     Adjusting scaling strategies to meet user service
       satisfaction;
20 else
21     Keep current scaling strategies;
```

4.2 | Lightweight SLO Resource Allocation Model

Microservices, owing to their loosely coupled structure, often result in user requests traversing one or more microservice chains to obtain the desired result. Each microservice within these chains operates at a distinct processing speed and may handle requests in parallel, each with its own latency. Ensuring that each microservice component consumes minimal system resources while meeting the user's RT requirements poses a significant optimization challenge. This gives rise to the following optimization problem:

$$\min \sum_{i}^{n} C_i \text{ subject to } \text{Tail\_latency}_k \leq SLO_k \tag{1}$$

where C_i denotes the total resources consumed by each microservice, Tail_latency_k denotes the eTL of each chain k, and SLO_k indicates a set RT constraint. The overarching optimization objective is to minimize the total amount of resources consumed by the microservices while ensuring the minimization of end-to-end latency violations. The symbols and definitions used in the paper are introduced in Table 2.

TABLE 2 | Symbols and definitions.

Symbols	Definitions
C_i	The total resources consumed by each microservices i
Tail_latency_k	The end-to-end tail latency of each chain k
SLO_k	Set SLO constraints for each microservice chain k
T_i	The service time for microservice i in a chain
τ	Zero-load latency of the microservice
λ	Arrival rate, typically defined as the number of requests arriving at the system per unit of time (per second)
μ	The maximum arrival rate that an instance of the microservice can sustain (without incurring infinite queuing)
ϕ	The shape of the LLP graph
s_i	SLO resources allocated to each microservice i
σ	The cost of a single instance of a microservice
ReC	The relative cost of a microservice
TC	The total cost for all microservice
pTL_i	Partial tail latency for each microservice i
eTL	End-to-end tail latency
$PSLO_i$	Partial SLO resource for each microservice i
x_i	Proportion of traffic flowing to each chain over a period of time i
O	Oscillation factor
OCT	Defined oscillation coefficient thresholds
CL_i	The load at a specific point in a time period
$\overline{CL}$	The average load value over the time period
E_i	The set of SLO resources assigned to microservice i during iterative computation
$\triangle ReC_i$	The change in relative cost per additional SLO block
d	Appropriate number of SLO blocks (in this paper we defined as 1000)
F	The functional model of the OCT

GrandSLAm [35] proposes to divide the end-to-end latency SLOs between microservices in proportion to their average service time (as shown in Equation (2)), but this approach does not fully optimize the reservation consumption and failure to meet the QoS needs of users.

$$\text{SLO}_k = \frac{T_k}{\sum_{i=1}^{n} T_i} \tag{2}$$

where T_i denotes the service time for microservice i in a chain.

Parslo proposes, based on the LLP graph (shown in Figure 2, with the function expression shown in Equation (3)) of microservice and the above model, the relationship between the relative total cost of microservice resource consumption and the allocation of the SLO resources to each microservice (as shown in Equation (4) and Figure 3). We define the σ parameter as the cost of a single microservice instance (like a pod in Kubernetes). We assume that all instances of a single microservice have the same cost without auto-scaling to change resources. When all microservices are deployed on the same type of instance, σ can be set to 1. However, since each microservice may be deployed on instances with different configurations (e.g., CPU cores, memory) or even different hardware capabilities, it is more realistic to optimize for the total cost of the microservices rather than simply considering the total number of instances. In the Figure 3, s represents the SLO resources allocated to the microservice.

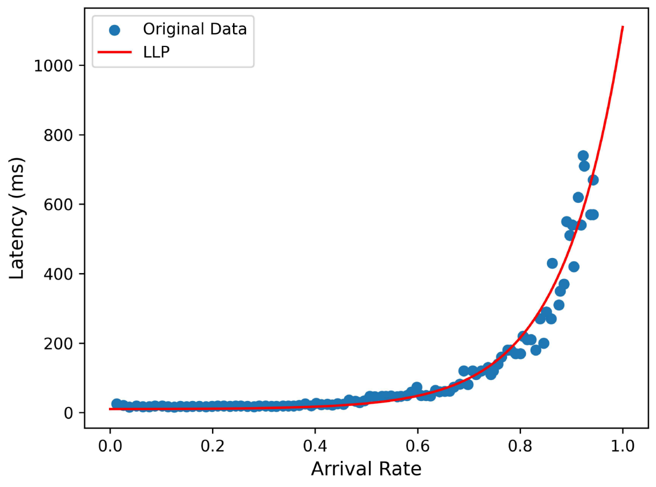


FIGURE 2 | LLP graph.

$$R = \tau\phi\left(\frac{\lambda}{\mu}\right) \tag{3}$$

where R denotes the RT of the microservice request, τ is the zero-load latency of the microservice, μ specifies the highest arrival rate that a microservice instance can handle without resulting in infinite queuing, λ is an arbitrary arrival rate, and the ϕ function is the shape of the LLP graph of a microservice instance.

$$ReC = \frac{\sigma}{\mu\phi^{-1}(\frac{s}{\tau})} \tag{4}$$

where ReC specifies the relative cost of a microservice, σ denotes the cost of a single instance of a microservice.

For a single chain with multiple microservices, Parslo assumes that when the SLO latency uses the end-to-end average latency as a metric, Equation (5) is used as a scale for SLO resource allocation to find the optimal SLO resource allocation scheme for each microservice, resulting in the least overall resource consumption.

$$\text{TC} = \sum_{i=1}^{n} ReC_i = \sum_{i=1}^{n} \frac{\sigma_i}{\mu_i\phi_i^{-1}(\frac{s_i}{\tau_i})}, \text{ with } \sum_{i=1}^{n} s_i = \text{SLO} \tag{5}$$

where TC denotes the total cost for all microservices.

However, when the end-to-end average latency is adopted as the SLO metric, it fails to fully capture the user's overall service satisfaction. Therefore, selecting the eTL as the SLO metric becomes necessary to ensure comprehensive user satisfaction. Nevertheless, using tail latency as the end-to-end SLO for resource allocation results in resource wastage, primarily due to tail latency being a non-accumulative metric. We have characterized the local tail latency incurred by different components processing service requests (SRs) on a microservice chain, as illustrated in Figure 4a. Here, SR1 represents the fastest service request processed among all requests, which also represents an optimal situation, showing the shortest time for each microservice to handle the requests. SR2 represents the slowest case of requests processing, with the longest processing time in each microservice component, serving to characterize the end of the tail latency for each component. SR3 represents the majority of requests contributing to end-to-end latency, where service requests are processed at normal speeds in some components but encounter longer processing times in one component, resulting in them being the slowest batch in terms of end-to-end latency. The quantitative relationship between these aspects is provided by Equation (6). The sum of partial tail latency (PTL) exceeds the eTL, leading to an excess of resources allocated to each microservice when eTL is utilized as the total SLO resource for allocation. Parslo proposes a combination of offline and online analytical models for addressing this issue, requiring significant time and computational resources for fitting in order to achieve a more satisfactory SLO resource allocation result.

$$\sum_{i=1}^{n} \text{pTL}_i \geq \text{eTL} \tag{6}$$

To reduce the consumption of computational and time resources and quickly obtain the optimized SLO resource allocation scheme, we model the relationship between the PTL for each microservice on a single chain and the eTL, as shown in Figure 4b. The relationship between the $\vec{\text{PTL}}$ and $\vec{\text{eTL}}$ as a sum of vectors is shown in Equation (7):

$$\sum_{i=1}^{n} \vec{\text{PTL}}_i = \vec{\text{eTL}} \tag{7}$$

Based on this abstract model, we use the $\vec{\text{eTL}}$ as a resource metric for SLO resource allocation and compute the allocation by the gradient descent method combined with the LLP of microservices to get the allocated SLO.

Through iterative optimization using gradient descent, we can find a nearly optimal partial SLO allocation scheme, thereby minimizing the deployment cost of end-to-end services. Although gradient descent typically guarantees convergence to a local optimum, as shown in Equation (5), the cost function we aim to optimize is convex. Therefore, we can use the gradient descent method to find the global optimal solution.

According to this, we construct vectors to derive Partial SLO ($\vec{\text{PSLO}}$), the value of which is defined in Equation (8) and closely approximates the actual $\vec{\text{eTL}}$. To simplify the calculation process,

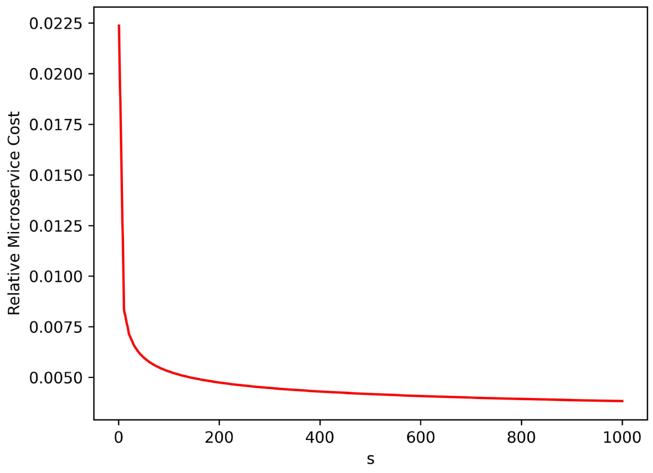


FIGURE 3 | Relative microservice cost.

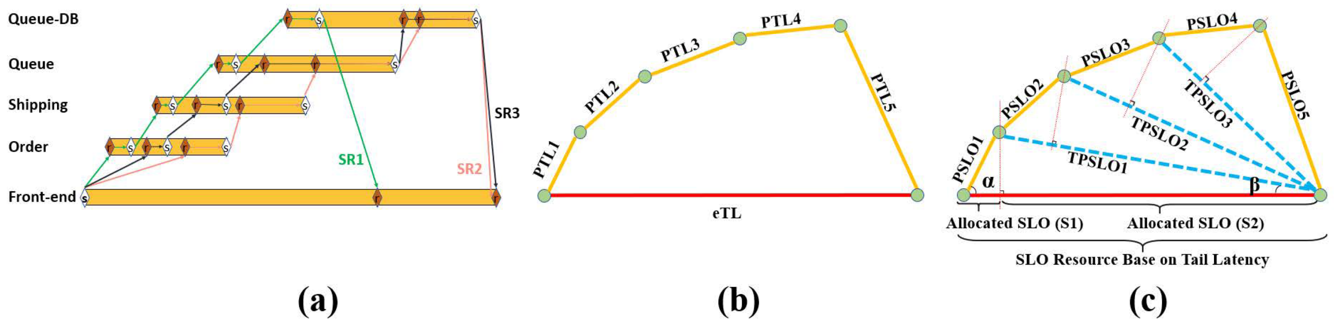


FIGURE 4 | Lightweight SLO allocation model.

we consolidate the latter nodes into one node (referred to as typical partial SLO, $\vec{\text{TPSLO}}$), as depicted in Figure 4c. However, the allocation process involves computing the SLO resources allocated to each node separately, ensuring that it does not impact the results of the SLO resource allocation.

$$\vec{\text{PSLO}}_1 = \frac{s_1}{\cos\alpha}, \quad \vec{\text{TPSLO}}_1 = \frac{s_2}{\cos\beta} \tag{8}$$

where $\vec{\text{PSLO}}_1$ indicates the amount of SLO resources to be allocated to microservice 1, s_1 denotes the assigned SLO value based on the eTL SLO calculation, and $\vec{\text{TPSLO}}_1$ means the sum of SLO resource vectors after $\vec{\text{PSLO}}_1$.

However, there exists no fixed relationship between pTL and eTL. Different types of microservices may exhibit various correlations, influenced by factors such as cluster state, internal communication, and hardware conditions. The mapping to the value of PSLO cannot be definitively determined, manifesting itself in the form of the angle between the vectors $\vec{\text{PSLO}}_1$ and $\vec{\text{PSLO}}_1$.

In order to maintain our lightweight and fast computation objective, we further simplify the model by setting the angle between the vectors PSLO_1 and TPSLO_1 to be a right angle. The point of intersection between them lies on the perpendicular line where the SLO assignment point is located. Consequently, we derive a lightweight solution to determine the PSLO values based on eTL SLO assignments, as shown in Equations (9)–(11):

$$\text{PSLO}_i = \sqrt{\text{SLO}_{end-to-end} * s_i} \tag{9}$$

$$s_i = s_0 + \sum_{b_j \in E_i} b_j \tag{10}$$

$$E_i = \{b_1 \dots b_j \cdots b_m\} \begin{cases} E_1 \cup E_2 \cup \cdots \cup E_i \cup \cdots \cup E_n = E \\ E_1 \cap E_2 \cap \cdots \cap E_i \cap \cdots \cap E_n = \emptyset \end{cases} \tag{11}$$

where s_i denotes the SLO resource allocated to each microservice i, s_0 represents a fixed value that is pre-assigned to each microservice, b_j denotes which round of iteration allocates SLO resources to microservice i, E_i indicates the set of SLO resources assigned to microservice. i during iterative computation, E denotes the total set of SLO resource iteration allocations.

In order to achieve the goal of a lightweight model and facilitate real-time computation, compared with the scheme of fitting to meet the optimal SLO resource allocation by constantly modifying the SLO resources, we choose this lightweight computation to obtain the optimized scheme, which can ensure the rapid adaptation of LSRAM in the environment of constant transformation of microservice applications and high-speed change of loads, which can not only ensure the QoS of the users but also save the resources.

4.3 | SLO Resource of Multi-Chain Shared Microservices and SLO Resources Update Algorithm

For complex chains that have typical structures such as dynamic branching, parallel fan-out, and microservice dependencies, similar to the end-to-end chains of Nested Fork Join DAGs [40], we adopt the approach of continuous merge into a single chain for calculation, so that each microservice chain with the same chain endpoint is finally merge into a single chain, in which the complex chain structure is abstracted into a microservice node (the merged process is shown in Figure 5).

During the resource allocation process, we partition the SLO resources into several equally-sized blocks. The size of these blocks must strike a balance; they should not be too large, as this could hinder the search for an optimal allocation scheme, nor too small, as this might excessively prolong the computation time. To balance the trade-offs, we allocate SLO resources ranging from 800 to 1200 SLO blocks, and each block is around 4 ms. Before the initial allocation phase, we distribute a specific number of SLO resource blocks to each microservice to ensure that, when employing the gradient descent method, it encounters a convergence point. This strategy helps prevent the allocation algorithm from continuously allocating resources to the same microservice indefinitely. Typically, LSRAM pre-allocates 10 SLO resource blocks (e.g., 40 ms) to each microservice. We then calculate the relative increase in resource consumption for each microservice along the single chain after adding a resource block. The change in resource consumption following the addition of each SLO block is illustrated in Equation (12). If there are cases where the SLO resources cannot be evenly divided by the SLO blocks, due to the large number of blocks we use, only a very small amount of SLO resources will generally remain. These remaining SLO resources will be allocated together to the microservice that saves the most resources during the final iteration.

$$\triangle ReC_i = \frac{\sigma_i}{\mu_i \phi_i^{-1}\left(\frac{s_i}{\tau_i}\right)} - \frac{\sigma_i}{\mu_i \phi_i^{-1}\left(\frac{s_i + \text{SLO}_{block}}{\tau_i}\right)} \tag{12}$$

where $\triangle ReC_i$ denotes the change in relative cost per additional SLO block.

After each microservice on a single chain increases by one resource block, we compute its relative increase in resource

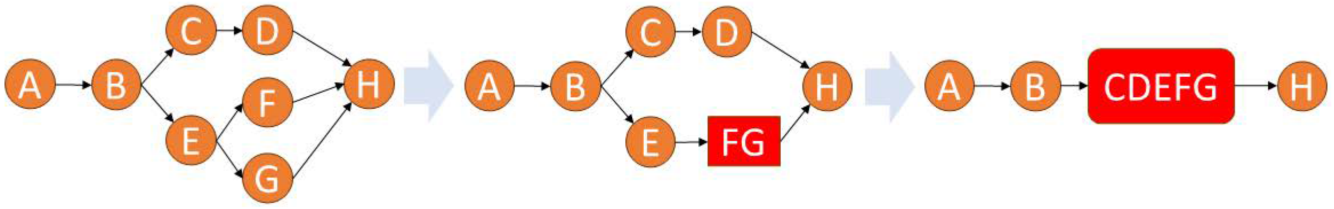


FIGURE 5 | Chains merge.

ALGORITHM 2 | SLO Resource Initialization Allocation Algorithm.

```
Input: SLO constraints for each microservice chain $SLO_k$,
       microservice load-latency-profile $\tau\phi\left(\frac{\lambda}{\mu}\right)$,
       microservice chain relationships $L$.
Output: SLO resources allocated to each microservice $s_i$
1  Initialize parameters:
2   SLO resource $SLO_{resource}$, appropriate number of blocks $d$,
    number of microservices $n$ in a chain, predefined SLO
    resource segments $SLO_{block}$, number of chains $k$, SLO
    resource $SLO_{shared}$ assigned to multi-chain shared
    microservices, the amount of change in ReC $\triangle ReC_i$ after
    each additional SLO block.
3  SLO Allocation:
4  $SLO_{block} = \frac{SLO_k}{d}$
5  $SLO_{resource} = SLO_k - (n * SLO_{block})$
6  if $SLO_{resource} \geq SLO_{block}$ then
7      $\triangle ReC_i = \frac{\sigma_i}{\mu_i \phi_i^{-1}(\frac{s_i}{\tau_i})} - \frac{\sigma_i}{\mu_i \phi_i^{-1}(\frac{s_i + SLO_{block}}{\tau_i})}$
8      Find the microservice with the largest $\triangle ReC_i$ in the
       queue;
9      Add SLO resources to this microservice.
10     $s_i = s_i + SLO_{block}$
11     $SLO_{resource} = SLO_{resource} - SLO_{block}$
12 $SLO_{shared} = \frac{\sum_{j=1}^{k} SLO_j}{k}$
```

consumption. Subsequently, we select the microservice with the least resource consumption. The evaluation criterion for the abstract node is the point with the least resource consumption increment in its status. The specific SLO resource allocation algorithm for the terminal of the same chain is depicted in Algorithm 2, where the SLO resource value assigned to the abstract node will actually be assigned to the point with the least incremental resource consumption within it.

In Algorithm 2, the inputs of the algorithm include SLO_k (end-to-end SLO constraints), the LLP graph function $\tau\phi\left(\frac{\lambda}{\mu}\right)$ for each microservice in a chain, and the relationship between the microservices in the chain (lines 1–3). Line 6 is the parameter definitions. To expedite computation, the algorithm divides the SLO resources into a number of SLO_{block} (line 8), which are then allocated one by one to the most resource-efficient microservices. Before starting the iterative allocation of SLO resources, it is necessary to allocate some SLO_{block} to each microservice in advance, crossing the point of the extreme value of ReC_i, to ensure that there is no negative number in the computation of $\triangle ReC_i$ (line 11). We utilize a queue to store the resource consumption of each microservice each time an SLO_{block} is allocated (line 12), ensuring that each update only involves the microservice node with the allocated SLO resources. Following SLO resource allocation, we initialize the SLO resources of the shared microservice (line 12).

Complexity Analysis: The time complexity of Algorithm 2 is $O(n \log n)$, where n represents the number of SLO_{block}; as in the algorithm, we use a queue to store the $\triangle ReC_i$ values after each calculation. In each iteration, only the SLO_{block} value of the microservice with the highest value from the previous round needs to be updated. The entire calculation process is only related to the number of SLO_{blocks}. Once the SLO_{blocks} have been distributed, the initialization of the entire SLO resources is completed; this meets our model's requirements for lightweight and fast computation.

However, managing microservices that are shared across multiple chains presents a challenge in determining the appropriate SLO resource allocation for each chain. This complexity arises in selecting the SLO resource value to use as a resource scheduling metric for the shared microservice. To overcome this problem, we conduct a detailed analysis of scenarios involving such shared microservices. In the initial allocation phase, we treat the shared microservice as unique and independent for each chain, as depicted in Figure 6 to illustrate replication separation. Based on this configuration, we compute the SLO resources allocated to the shared microservice at each chain endpoint. During the initial allocation process, we adopt the method showen in Equation (13) to obtain an initial allocation value for the SLO resources.

$$\text{SLO}_{shared} = \frac{\sum_{j=1}^{k} \text{SLO}_j}{k} \tag{13}$$

$$\text{SLO}_{shared} = \sum_{j=1}^{k} x_j * \text{SLO}_j \tag{14}$$

where k denotes the number of microservices in a chain, x_j denotes the proportion of traffic for any microservice in a chain, and SLO_{shared} indicates the SLO resources that should be allocated to the multi-chain shared microservice.

In the actual production environment, the loads distributed to different chains within a period of time are inconsistent. Figure 6c, in order to ensure that our model can continuously adapt to different production environments, we design an updated algorithm for LSRAM to allocate SLO resources for shared microservices, which lies in the fact that the LSRAM collects the load traffic flow of the shared microservices flowing to different end-to-end services within a period of time. LSRAM's SLO resource allocation update algorithm still follows the principle of lightweight and can adapt to changes in the load flow direction of the cluster Equation (14), so that the cluster scheduling can ensure user service satisfaction. LSRAM's SLO resource update allocation algorithm still follows the principle of lightweight and adapts to changes in the load flow of the cluster, enabling cluster scheduling to ensure user service satisfaction. When the microservice application, attribute, or instance specification in the cluster changes, LSRAM will reallocate the microservice's SLO resources. Our SLO resource update allocation algorithm is shown in Algorithm 3. Lines 1 to 3 are the inputs requested by the algorithm; line 5 is the settings of the parameters. When there is a change in the cluster environment (including changes in system resource capacity, changes in cluster hardware, redeployment of containers, etc.) or a pre-set time is reached, the algorithm will reallocate the SLO resources of the microservices (line 10) and compute the SLO resources of the shared microservices according to line 11.

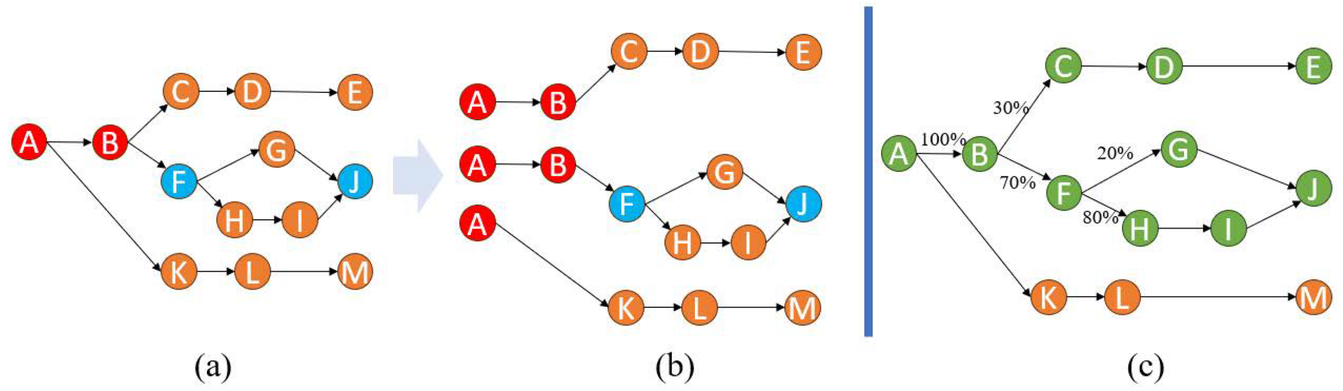


FIGURE 6 | (a) denotes the chain structure of the microservice; (b) indicates the splitting of the microservice chain architecture to obtain multiple single chains; (c) specifies the proportion of traffic going to different chains over a period of time.

ALGORITHM 3 | SLO Resource Update Allocation Algorithm.

```
   Input: Proportion of traffic distributed to each chain over a
          period of time $x_i$, initial allocation of SLO resources
          to each microservice $s_i$, microservice chain
          relationships $L$.
   Output: SLO resources allocated to each microservice $s_i$
1  Initialize parameters: SLO resource $SLO_{shared}$ assigned
   to multi-chain shared microservice, elapsed time $t$,
   predefined resource reallocation time thresholds $M_t$, SLO
   resource $SLO_{new}$ of new microservice.
2  Update:
3  if $t \geq M_t$ or cluster environment changes then
4      $SLO_{shared} = \sum_{j=1}^{k} x_j * SLO_j$
5      Reallocating SLO resources for microservices $s_i$
6      if cluster environment changes then
7          Collect the LLP graphs for new microservices $i$
8          $s_i = s_0 + \sum_{b_j \in E_i} b_j$
9          if New microservice connects to shared microservice
           then
10             $SLO_{shared} = SLO_{shared} + \frac{1}{k} SLO_{new}$
```

4.4 | Esdnn Prediction Model and Mode Converter

To enhance the LSRAM's ability to cope with highly fluctuating loads and bursty loads, we utilize the esDNN [39] load prediction model and design the mode converter. The esDNN model is our previous work, which is an improved model based on Gated Recurrent Unit (GRU). It's lightweight and straightforward, enabling quick predictions of the load value for the next time slot.

We trained our prediction model using Alibaba's dataset [41] for updating, focusing on high volatile changes in load and burst load for intensive training. The training was also targeted on the load of different microservices, which ensures the accuracy of the prediction model in predicting the load of different microservices for the next time slice.

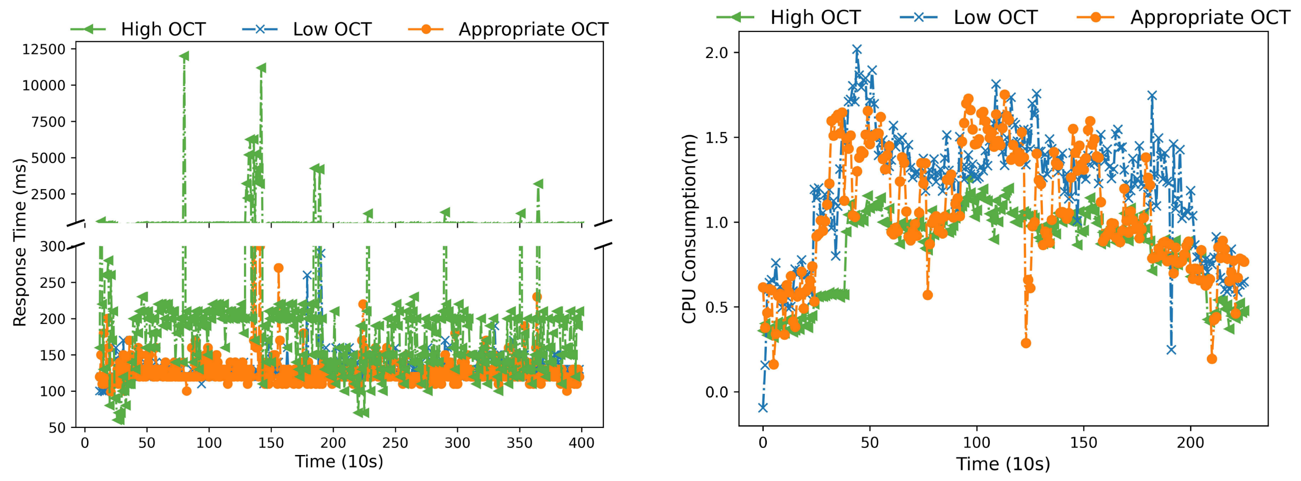
To ensure the QoS of users to the maximum extent under high fluctuating loads, we design a mode converter for high fluctuating loads, and when the oscillation-aware function in the mode converter detects that the system is in a state of high fluctuating loads, it will change the resource scaling policy of the resource scheduler to ensure the QoS of users. We use a sliding window mechanism to store past load information and detect if the load is oscillating by calculating the variance of the load relative to the average load. A high O value indicates significant load fluctuations, while a low O value indicates more stable load conditions. The oscillation-aware function is shown in Equation (15):

$$O = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}[CL_i - \overline{CL}]^2}}{\overline{CL}} \tag{15}$$

where O denotes the oscillation factor, CL_i denotes the load at a specific point in a time period, and $\overline{CL}$ indicates the average load value over the time period.

An appropriate oscillation coefficient threshold (OCT) is crucial for the mode converter. If the oscillation coefficient is set too high, the converter cannot detect system fluctuations, failing to switch modes promptly, which results in severe SLO violations in microservices. Conversely, if the oscillation coefficient is set too low, the system frequently switches to a high oscillation mode, leading to resource wastage. As illustrated in the Figure 7, we have demonstrated the impact of different oscillation coefficient thresholds on response times and resource utilization. We can observe that when the OCT is set too high, the system becomes insensitive to fluctuating loads, resulting in serious SLO violations. On the other hand, when the OCT is set too low, the system becomes overly sensitive to load fluctuations, causing the frequent resource scaling operations. Although this effectively ensures service quality, it leads to significant resource waste.

Moreover, different microservices exhibit varying sensitivities to load fluctuations, so the OCT should be set accordingly for each microservice. Through analysis and synthesis, we found that the OCT setting is related to the LLP parameters of the microservices and the partial SLO assigned to them. The steeper the LLP curve of a microservice, the more sensitive it is to load changes. When the same number of requests is sent to microservices with different LLP steepness, the steeper ones experience a faster increase



(a) The impact of different OCTs on response time under the same load conditions. **(b)** CPU consumption of microservices under different OCTs.

FIGURE 7 | Performance under different OCTs. (a) The impact of different OCTs on response time under the same load conditions. (b) CPU consumption of microservices under different OCTs.

in response time, making them prone to request timeouts and service crashes when receiving fluctuating loads. Additionally, the more partial SLO assigned to a microservice, the more susceptible it becomes to fluctuating loads, meaning that it requires more SLO time to process the same number of requests. Based on this, we modeled the relationship between these three factors and derived the Equation (16).

$$\mathrm{OCT} = \left(F \left(\frac{(\mathrm{PSLO})^2}{\mu \phi^{-1} \left(\frac{\mathrm{PSLO}}{\tau} \right)} \right) \right) \tag{16}$$

where F denotes the functional model of the OCT.

5 | Experimental Evaluations

In this section, we present information about our experiments, including the environment setup for the experiment, the datasets used, and the application. Secondly, we will demonstrate and analyze the results of our experiments and explain the improvement of our methodology.

5.1 | Experimental Setup

Cluster Configuration: All of our experiments are validated on a prototype system with the following server configurations: There are a total of 10 nodes, comprising 3 master nodes and 7 worker nodes. The 3 master nodes are equipped with 32 cores and 64GB of memory each; among the 7 worker nodes, 4 are configured with 32 cores and 64GB of memory, and the remaining worker nodes are with 56 cores and 128GB memory, 104 cores and 256GB of memory, and 64 cores and 64GB of memory. The microservices were deployed on top of Kubernetes.

Datasets: To verify the effectiveness of our method in a realistic environment, we employed the dataset obtained from Alibaba Cluster [41], which was sourced from production clusters within Alibaba encompassing over ten thousand bare-metal nodes during a 13-day period in 2022. The dataset records the CPU and memory utilization of over 470,000 containers spanning more than 28,000 microservices within the same production cluster. We transform the data from this dataset into request counts and generate workloads using Locust[3], which was configured to mimic application requests for stress tests. We also convert the fluctuations in CPU utilization on the server to changes in the number of requests sent by Locust by measuring the relationship between container CPU utilization and load requests in the Alibaba dataset.

Tested Application: Sock-shop [42] is an e-commerce website microservice whose functions include searching, ordering, and shipping, which can be divided into three parts: front-end, business logic, and database, and contains a total of 13 microservices. We conducted breadth load testing on the microservice components of Sock-shop's critical chains and carved out its LLP graph in detail, as shown in the Figure 9.

5.2 | Baseline Approach

To evaluate the performance of LSRAM, we compare it with several state-of-the-art baselines as follows:

Horizontal Pod Autoscaling (HPA) [31]: It is a heuristic autoscaling framework based on a predefined threshold, which uses CPU utilization as its resource scaling metric in this experiment and automatically scales resources when the system detects that the CPU utilization of an instance exceeds a fixed threshold. The threshold is set to [30%, 50%] in this experiment, as this value has been used in [43].

Fuzzy-Based Auto-scaler (FuSys) [19]: It is a heuristic dynamic threshold scheduling framework based on fuzzy control theory. Fuzzy will dynamically adjust the system's resource scheduling thresholds based on the system's current resource usage and load by fuzzy control.

Parslo [36]: It is a method to compute the near-optimal SLO resource allocation scheme based on the gradient descent method; each microservice will perform heuristic resource scaling based on this scheme.

5.3 | Evaluations

We evaluate the proposed LSRAM with the above baselines in the same experimental settings. To avoid the randomness of experiments, we repeat each baseline 5 times and compute its average value to compare performance.

The tests comparing LSRAM with other baselines clearly demonstrate its significant improvement on resource savings, as depicted in the Figure 10a. LSRAM consistently consumes the minimum amount of system resources under the same load pressure variations. It maintains a relatively stable RT while minimizing CPU resource consumption. The scaling threshold of HPA was set with [30%, 50%] CPU utilization to ensure RT, resulting in the highest resource consumption. FuSys saves resources compared with HPA but experienced significant RT fluctuations. The reason why LSRAM and Parslo have relatively similar resource consumption in the early stages is that they both have near-optimal initial SLO resource allocation schemes. However, as the system keeps running, LSRAM continuously adapts to the current environment through its SLO resource updating algorithm, leading to a reduction in overall resource consumption.

To further demonstrate LSRAM's ability to maintain stable response times during runtime. With numerical analysis for RT, we can note that the Cumulative Distribution Function (CDF) graph depicted in Figure 11a shows that LSRAM's 20% RT is around 190 ms, and 80% of the RT are less than 250 ms, with the maximum value of 260 ms. Although 80% of the RT of the other three methods are around 200 ms, all of them have exceptionally maximum RT (tail latency), e.g., the tail latency of FuSys is around 400 ms. LSRAM maintains the RT within the acceptable SLO, which ensures the QoS and saves resources compared with the other three baselines. The main reason is that LSRAM constantly adapts to the current environment and adjusts the resource allocation threshold through the prediction model and the SLO resource update model.

We also further analyze the total resources consumed of each baseline; LSRAM has saved the total resource consumption by 30% compared with HPA, 21.4% compared with FuSys, and 17% compared with Parslo, as shown in Figure 11b. LSRAM's lightweight SLO allocation model and SLO resource update algorithm enable the framework to compute SLO resource schemes that match the current cluster environment in real-time without significant resource consumption. The SLO resource update algorithm dynamically adjusts the allocation scheme based on the historical load distribution and microservice characteristics. This ensures that the overall SLO resource scheme continuously remains aligned with the current cluster environment, which is the main reason that LSRAM is capable of maintaining low resource consumption.

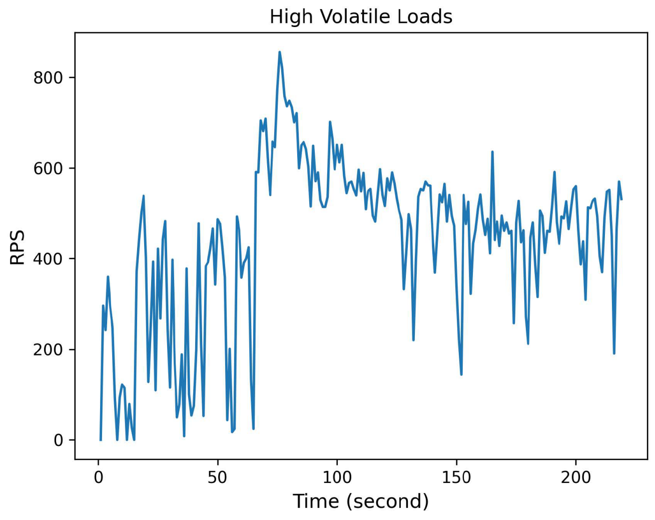


FIGURE 8 | The highly volatile load.

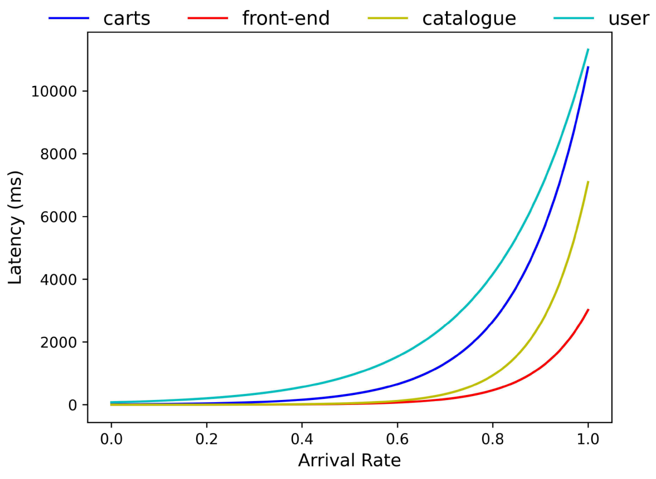
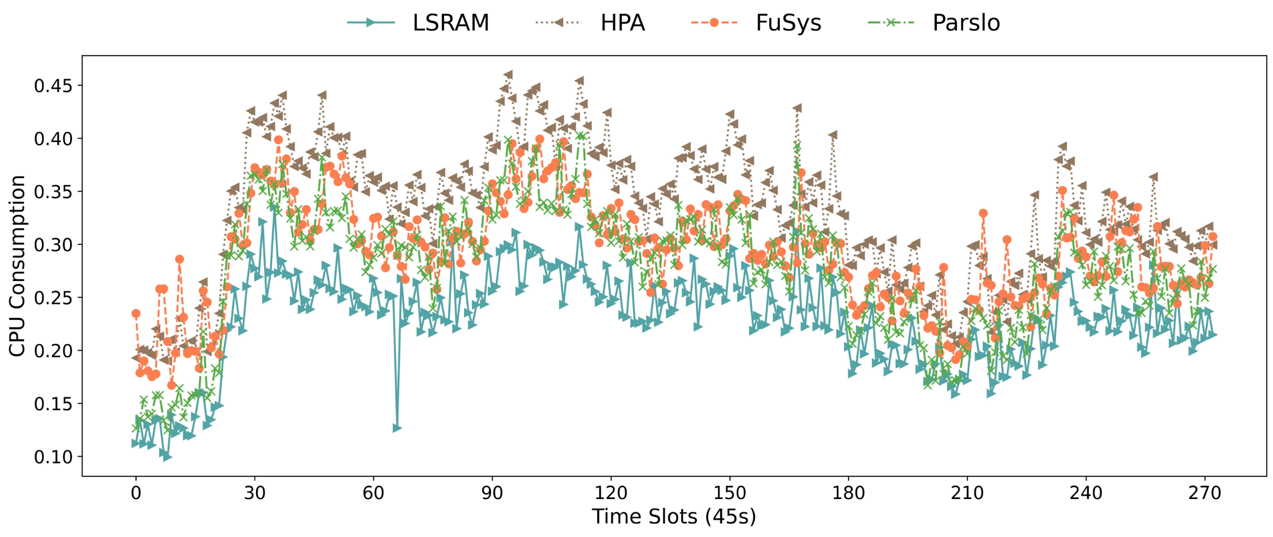


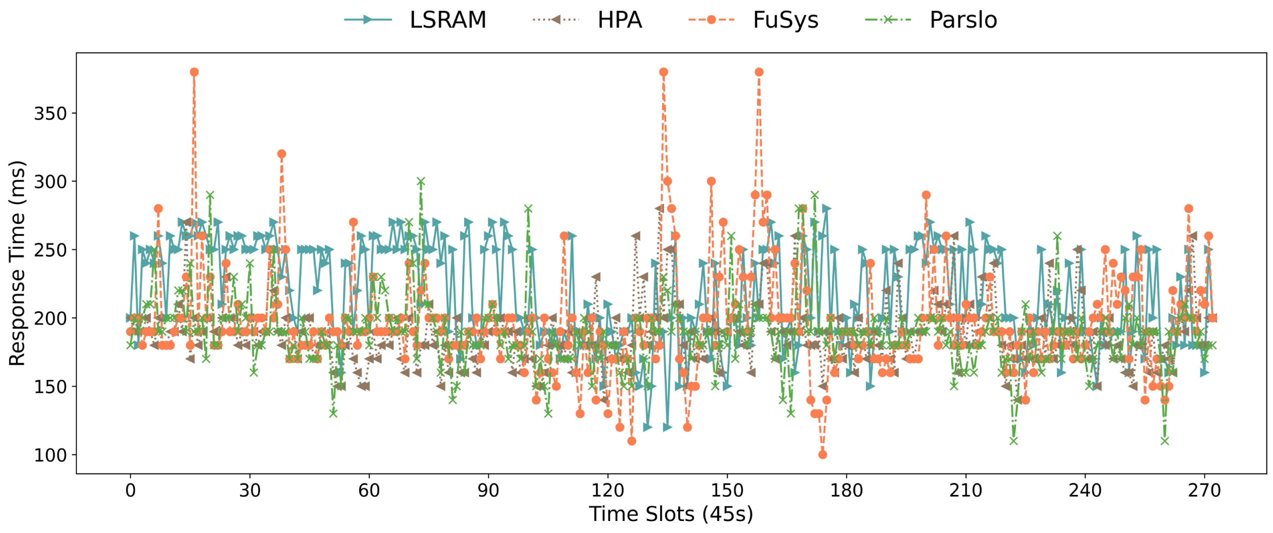
FIGURE 9 | LLP graphs of Sock-shop's critical microservices.

To measure LSRAM's performance under highly volatile load scenarios, we selected a steep load data pattern, as illustrated in the Figure 8 with a maximum difference between peak and bottom can be 700 requests per second (RPS). The results demonstrate that LSRAM effectively maintains users' QoS, which keeps the RT to be stable at 250 ms, with a maximum RT of 400 ms under extreme high loads. Parslo and FuSys are more sensitive to violating loads, resulting in congested request processing. HPA, due to its heuristic scaling and fixed threshold, experiences severe SLO violations when there are spikes.

In contrast, the other baselines, due to their heuristic nature, struggle to allocate resources promptly to handle such load scenarios, resulting in a significant increase in RT, as depicted in Figure 12. During runtime, LSRAM's load predictor accurately forecast load changes in future time periods as we have trained our load prediction model to fit with bursts. However, the other baselines respond heuristically to current load changes, often failing to provide timely responses to large fluctuation ranges in loads, even leading to instances of crashes. In addition to load

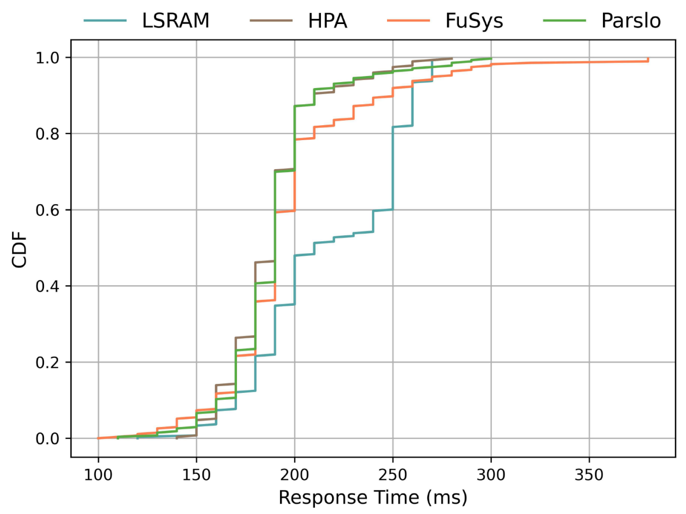


(a) CPU Resource Consumption.

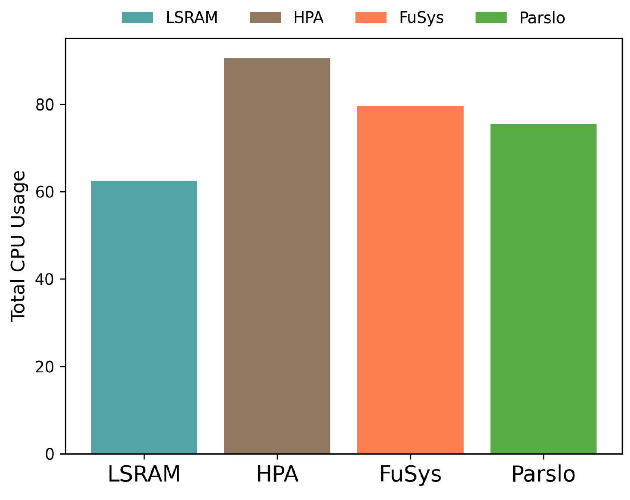


(b) Response Time (P99).

FIGURE 10 | Performance under the same load. (a) CPU resource consumption. (b) Response time (P99).



(a) CDF for Response Time.



(b) Total CPU.

FIGURE 11 | Analysis of results, (a) CDF for response time. (b) Total CPU.

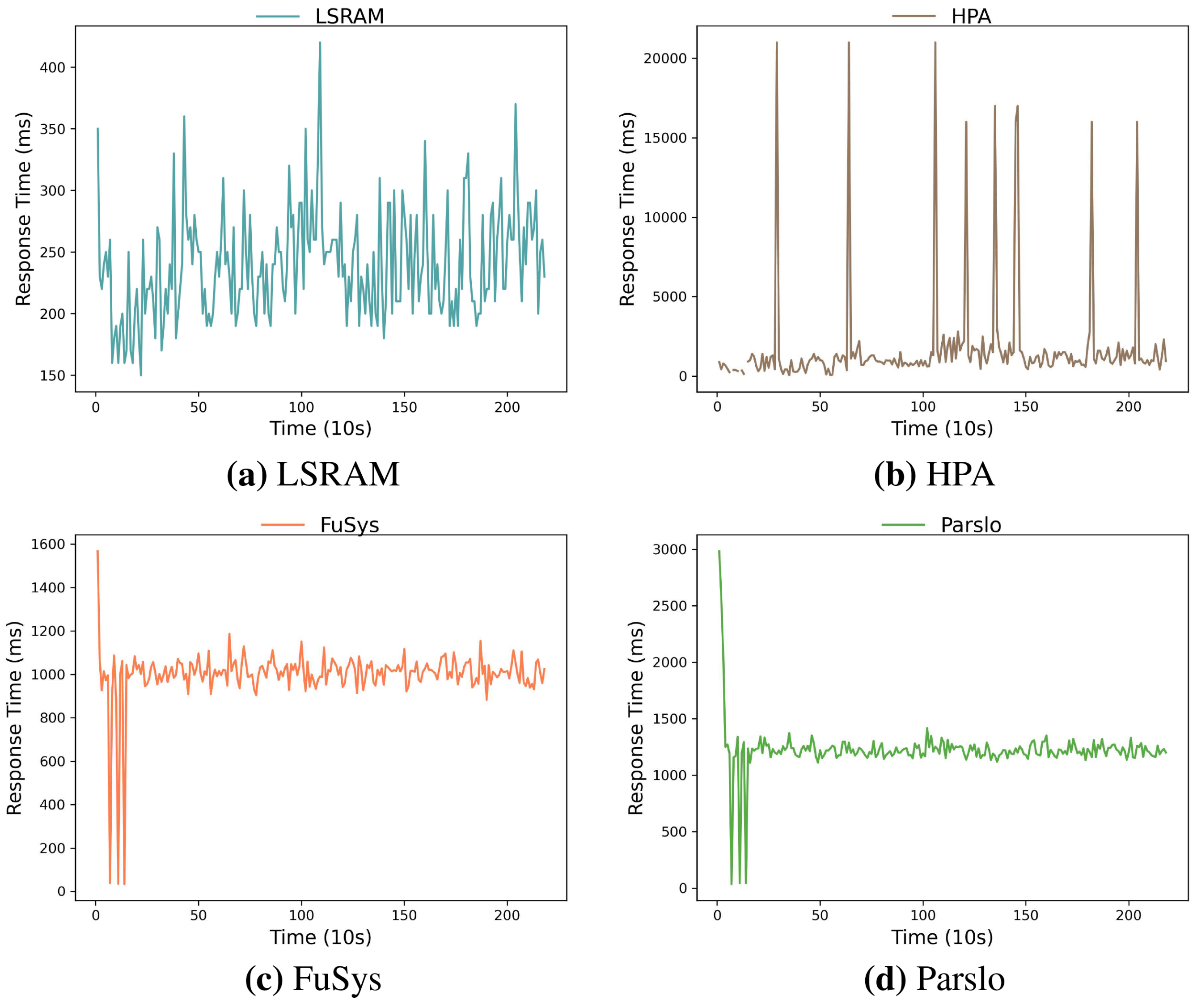


FIGURE 12 | Performance under Highly Volatile Load. (a) LSRAM, (b) HPA, (c) FuSys, (d) Parslo.

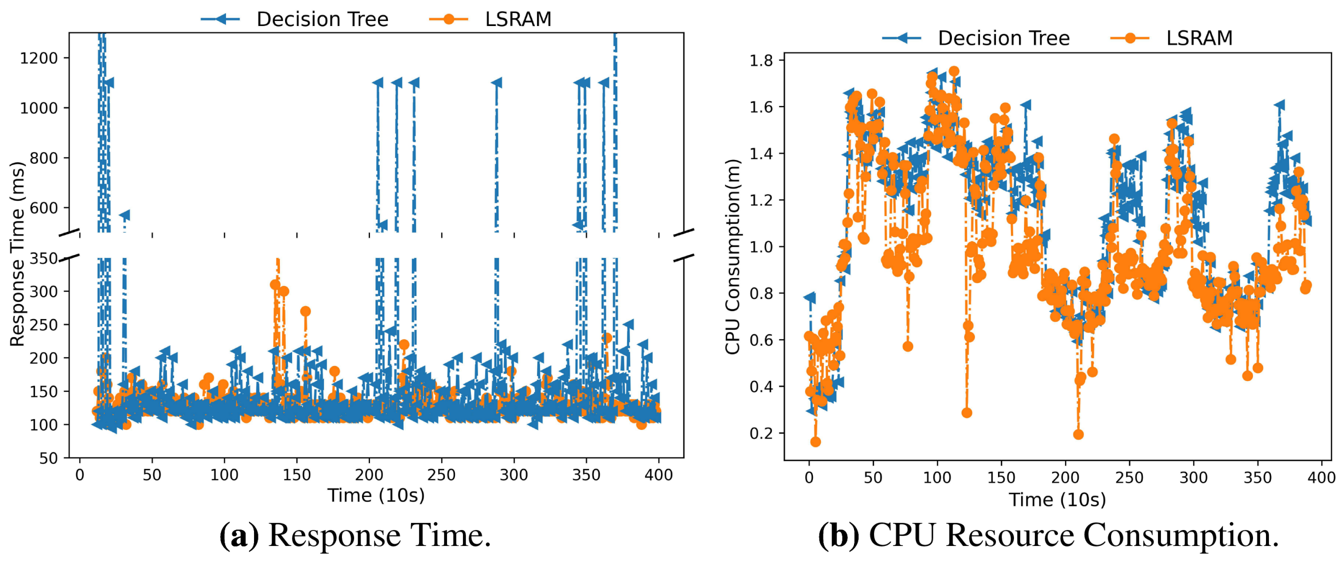


FIGURE 13 | Comparison between different mode converters. (a) Response Time. (b) CPU Resource Consumption.

prediction, LSRAM incorporates a mode converter that monitors load changes over time. When detecting load oscillations, LSRAM adopts a scaling mechanism to maintain SLO. This proactive approach further enhances LSRAM's ability to handle highly volatile loads effectively.

To validate the effectiveness of our mode converter, we compared it with the current mainstream mode converter. We compared our model converter with a decision tree-based pattern converter [44]. The decision tree's depth is primarily determined by variance features, load variation trends, and the gap between maximum and minimum values. As shown in the experimental results in Figure 13, LSRAM outperforms the decision tree-based pattern converter in ensuring SLO compliance, not only providing better QoS but also conserving more system resources. During testing, the decision tree-based converter occasionally misjudged or delayed decisions, causing it to fail to switch modes promptly during certain fluctuating traffic periods, resulting in a spike in service request response times. When comparing CPU consumption, as shown in Figure 13b, the decision tree converter also exhibited misjudgment during operation, treating non-significant load fluctuations as significant ones and triggering mode switching, leading to increased CPU resource consumption during certain periods.

6 | Conclusions and Future Work

In this paper, we identify limitations in the current SLO resource allocation-based autoscaling framework regarding the accuracy of allocation time duration calculation and managing unexpected traffic flow. Consequently, we propose a lightweight SLO resource allocation model to swiftly compute the allocation scheme. Additionally, we introduce an update model enabling LSRAM to adjust microservices' SLO resources based on real-time cluster statuses. To handle bursty traffic and fluctuating loads effectively, we enhance the esDNN prediction model and introduce a pattern converter. Our experimental evaluation in a clustered environment demonstrates that LSRAM not only ensures effective QoS but also reduces resource usage by 17% compared to the state-of-the-art baseline approach.

In future work, we plan to explore the relationship between end-to-end latency and partial tail latency using deep learning methods. Specifically, we intend to explore more complex and extensive microservices architectures, where multiple tasks are dynamically scheduled and managed. This includes scenarios involving heterogeneous workloads and mixed workload patterns. Furthermore, we hope to investigate a reinforcement learning-based SLO resource allocation architecture that can quickly adapt to load and environmental changes, thereby improving system resource utilization efficiency.

Author Contributions

All authors contributed to the final manuscript by discussing the ideas and analyzing the results. Minxian Xu proposed the initial idea and guided the experimental design and manuscript writing. Kan Hu optimized the initial idea, designed and conducted the experiments, and wrote the main part of this work. Kejiang Ye and Chengzhong Xu provided insights on the suitable scenario and the scalability of the proposed approach and improved the writing of this manuscript.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62072451, 62102408, 92267105), Guangdong Basic and Applied Basic Research Foundation (No. 2024A1515010251, 2023B1515130002), Guangdong Special Support Plan (No. 2021TQ06X990), Shenzhen Basic Research Program under grants JCYJ20220818101610023, and JCYJ20240809180935001, Shenzhen Industrial Application Projects of undertaking the National key R & D Program of China (No. CJGJZD20210408091600002).

Data Availability Statement

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

Endnotes

[1] https://prometheus.io/.

[2] https://www.jaegertracing.io/.

[3] https://locust.io/.

References

1. H. Zeng, T. Wang, A. Li, Y. Wu, H. Wu, and W. Zhang, "Topology-Aware Self-Adaptive Resource Provisioning for Microservices," *in 2023 IEEE International Conference on Web Services* (2023): 28–35.

2. Y. Gan, Y. Zhang, D. Cheng, et al., "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," *in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019): 3–18.

3. Amazon, "Microservices at Amazon," 2015, https://www.slideshare.net/apigee/i-love-apis2015-microservices-at-amazon-54487258.

4. Netflix Techblog, "Microservices - Netflix Techblog," 2021, https://netflixtechblog.com/tagged/microservices.

5. S. Luo, H. Xu, C. Lu, et al., "Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis," *in Proceedings of the ACM Symposium on Cloud Computing* (2021): 412–426.

6. M. Xu, L. Yang, Y. Wang, et al., "Practice of Alibaba Cloud on Elastic Resource Provisioning for Large-Scale Microservices Cluster," *Software: Practice and Experience* 54, no. 1 (2024): 39–57.

7. S. Luo, H. Xu, K. Ye, et al., "ERMS: Efficient Resource Management for Shared Microservices With SLA Guarantees," *in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022): 62–77.

8. Amazon, "Amazon found every 100ms of latency cost them 1% in sales," 2023, https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales.

9. C. Delimitrou and C. Kozyrakis, "Amdahl's Law for Tail Latency," *Communications of the ACM* 61, no. 8 (2018): 65–72.

10. S. N. A. Jawaddi, M. H. Johari, and A. Ismail, "A Review of Microservices Autoscaling With Formal Verification Perspective," *Software: Practice and Experience* 52, no. 11 (2022): 2476–2495.

11. A. Mirhosseini and T. F. Wenisch, "The Queuing-First Approach for Tail Management of Interactive Services," *IEEE Micro* 39, no. 4 (2019): 55–64.

12. B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile Dynamic Provisioning of Multi-Tier Internet Applications," *ACM Transactions on Autonomous and Adaptive Systems* 3, no. 1 (2008): 1–39.

13. R. Khorsand, M. Ghobaei-Arani, and M. Ramezanpour, "FAHP Approach for Autonomic Resource Provisioning of Multitier Applications in Cloud Computing Environments," *Software: Practice and Experience* 48, no. 12 (2018): 2147–2173.

14. S. Hassan, R. Bahsoon, and R. Buyya, "Systematic Scalability Analysis for Microservices Granularity Adaptation Design Decisions," *Software: Practice and Experience* 52, no. 6 (2022): 1378–1401.

15. L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes Controller for Managing the Availability of Elastic Microservice Based Stateful Applications," *Journal of Systems and Software* 175 (2021): 110924.

16. S. N. Srirama, M. Adhikari, and S. Paul, "Application Deployment Using Containers With Auto-Scaling for Microservices in Cloud Environment," *Journal of Network and Computer Applications* 160 (2020): 102629.

17. Kubernetes Horizontal Pod Autoscaler. 2021, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

18. K. Rzadca, P. Findeisen, J. Swiderski, et al., "Autopilot: Workload Autoscaling at Google," in Proceedings of the Fifteenth European Conference on Computer Systems (2020): 1–16.

19. B. Liu, R. Buyya, and T. A. Nadjaran, "A Fuzzy-Based Auto-Scaler for Web Applications in Cloud Computing Environments," in *Service-Oriented Computing: 16th International Conference, ICSOC 2018* (Hangzhou, China: November 12-15, 2018, Proceedings 16: Springer, 2018), 797–811.

20. Y. Song, C. Li, K. Zhuang, T. Ma, and T. Wo, "An Automatic Scaling System for Online Application With Microservices Architecture," in *2022 IEEE International Conference on Joint Cloud Computing, JCC 2022* (Fremont, CA: IEEE, 2022), 73–78.

21. Y. Li, H. Zhang, W. Tian, and H. Ma, "Joint Optimization of Auto-Scaling and Adaptive Service Placement in Edge Computing," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems, ICPADS 2021* (Beijing, China: IEEE, 2021), 923–930.

22. Y. Gan, Y. Zhang, K. Hu, et al., "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices," in *in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019): 19–33.

23. Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "SAGE: Practical and Scalable ML-Driven Performance Debugging in Microservices," *in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021): 135–151.

24. X. Yu, W. Wu, and Y. Wang, "Dependable Workflow Scheduling for Microservice QoS Based on Deep Q-Network," in *2022 IEEE International Conference on Web Services, ICWS 2022* (Barcelona, Spain: IEEE, 2022), 240–245.

25. X. Chen, L. Yang, Z. Chen, G. Min, X. Zheng, and C. Rong, "Resource Allocation With Workload-Time Windows for Cloud-Based Software Services: A Deep Reinforcement Learning Approach," *IEEE Transactions on Cloud Computing* 11, no. 2 (2022): 1871–1885.

26. Z. Jian, X. Xie, Y. Fang, et al., "DRS: A Deep Reinforcement Learning Enhanced Kubernetes Scheduler for Microservice-Based System," *Software: Practice and Experience.* 54, no. 10 (2024): 2102–2126.

27. M. Xu, C. Song, S. Ilager, et al., "CoScal: Multifaceted Scaling of Microservices With Reinforcement Learning," *IEEE Transactions on Network and Service Management* 19, no. 4 (2022): 3995–4009.

28. B. Cai, B. Wang, M. Yang, and Q. Guo, "AutoMan: Resource-Efficient Provisioning With Tail Latency Guarantees for Microservices," *Future Generation Computer Systems* 143 (2023): 61–75.

29. 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 14th USENIX Symposium on Operating Systems Design and Implementation* (2020): 805–825.

30. 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 the 17th International Conference on Emerging Networking Experiments and Technologies* (2021): 154–167.

31. C. Qu, R. N. Calheiros, and R. Buyya, "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey," *ACM Computing Surveys* 51, no. 4 (2018): 1–33.

32. M. H. Fourati, S. Marzouk, and M. Jmaiel, "Epma: Elastic Platform for Microservices-Based Applications: Towards Optimal Resource Elasticity," *Journal of Grid Computing* 20, no. 1 (2022): 6.

33. M. Yan, X. Liang, Z. Lu, J. Wu, and W. Zhang, "HANSEL: Adaptive Horizontal Scaling of Microservices Using bi-LSTM," *Applied Soft Computing* 105 (2021): 107216.

34. A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," *In: Proceedings of the ACM Symposium on Cloud Computing*. 2021: 427–441.

35. 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 the Fourteenth EuroSys Conference* (2019): 1–16.

36. 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 the ACM Symposium on Cloud Computing* (2021): 442–457.

37. A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems* (Texas: IEEE, 2019), 1994–2004.

38. L. Zhao, Y. Yang, K. Zhang, et al., "Rhythm: Component-Distinguishable Workload Deployment in Datacenters," *in Proceedings of the Fifteenth European Conference on Computer Systems* (2020): 1–17.

39. M. Xu, C. Song, H. Wu, S. S. Gill, K. Ye, and C. Xu, "esDNN: Deep Neural Network Based Multivariate Workload Prediction in Cloud Computing Environments," *ACM Transactions on Internet Technology* 22, no. 3 (2022): 1–24.

40. J. Fonseca, G. Nelissen, and V. Nélis, "Schedulability Analysis of DAG Tasks With Arbitrary Deadlines Under Global Fixed-Priority Scheduling," *Real-Time Systems* 55 (2019): 387–432.

41. Alibaba, "Alibaba Cluster Trace Dataset: Microservices Version 2022," 2022, https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022.

42. Sock shop microservice demo application. 2016, https://microservices-demo.github.io.

43. H. Ahmad, C. Treude, M. Wagner, and C. Szabo, "Smart HPA: A Resource-Efficient Horizontal Pod Auto-Scaler for Microservice Architectures," *in 21st IEEE International Conference on Software Architecture* (2024).

44. X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo, "ANT-Man: Towards Agile Power Management in the Microservice Era," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (Georgia: IEEE, 2020), 1–14.