Sora: A Latency Sensitive Approach for Microservice Soft Resource Adaptation

Jianshu Liu
Louisiana State University
Baton Rouge, USA
jliu96@lsu.edu

Qingyang Wang
Louisiana State University
Baton Rouge, USA
qwang26@lsu.edu

Shungeng Zhang
Augusta University
Augusta, USA
szhang2@augusta.edu

Liting Hu
University of California Santa Cruz
Santa Cruz, USA
liting@ucsc.edu

Dilma Da Silva
Texas A&M University
College Station, USA
dilma@cse.tamu.edu

ABSTRACT

Fast response time for modern web services that include numerous distributed and lightweight microservices becomes increasingly important due to its business impact. While hardware-only resource scaling approaches (e.g., FIRM [47] and PARSLO [40]) have been proposed to mitigate response time fluctuations on critical microservices, the re-adaptation of soft resources (e.g., threads or connections) that control the concurrency of hardware resource usage has been largely ignored. This paper shows that the soft resource adaptation of critical microservices has a significant impact on system scalability because either under- or over-allocation of soft resources can lead to inefficient usage of underlying hardware resources. We present Sora, an intelligent, fast soft resource adaptation management framework for quickly identifying and adjusting the optimal concurrency level of critical microservices to mitigate service-level objective (SLO) violations. Sora leverages online fine-grained system metrics and the propagated deadline along the critical path of request execution to quickly and accurately provide optimal concurrency setting for critical microservices. Based on six real-world bursty workload traces and two representative microservices benchmarks (*Sock Shop* and *Social Network*), our experimental results show that Sora can effectively mitigate large response time fluctuations and reduce the 99th percentile latency by up to 2.5× compared to the hardware-only scaling strategy FIRM [47] and 1.5× to the state-of-the-art concurrency-aware system scaling strategy ConScale.

CCS CONCEPTS

• **General and reference → Performance**; **Experimentation**; • **Computer systems organization → Cloud computing**.

KEYWORDS

Scalability, Microservices, Auto-scaling, Soft Resource

ACM Reference Format:
Jianshu Liu, Qingyang Wang, Shungeng Zhang, Liting Hu, and Dilma Da Silva. 2023. Sora: A Latency Sensitive Approach for Microservice Soft Resource Adaptation. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3590140.3592851

1 INTRODUCTION

In recent years, modern user-facing applications have witnessed a wide adoption of microservices-based architecture. Many industry practitioners, such as Twitter [1], Netflix [5], and Alibaba [36], have migrated their applications from the classic monolithic design to microservices. An important reason is that the microservices-based architecture can decouple an application into tens or hundreds of loosely-coupled microservices to provide superior scalability. The scalable fine-grained component microservices [46] can provide greater performance control by only adding system resources to the components needing extra capacity. However, achieving efficient resource scaling for microservices is challenging since the applications must meet stringent Service-Level Objectives (SLOs) like bounded response time while handling the naturally bursty workload. For example, Amazon found that every 100ms of latency cost them 1% in sales when facing 20× normal-traffic of peak load over holidays (e.g., Black Friday) [15].

A recent study [47] explored the hardware-only resource scaling approach for the critical microservice instances to handle the bursty workload. A significant insight is that the transient contention of low-level shared hardware resources such as caches and memory is the major contributing factor to service-level objectives (SLOs) violation. However, they barely discuss the complex soft resource (e.g., threads or database connections) re-adaptation of these microservices to match the hardware resource changes after the system scaling, which has been shown to impact the overall system performance significantly [33, 64]. For example, Figure 1 shows that the hardware-only Kubernetes Horizontal Pod Autoscaling [7] cannot reduce the response time spikes due to over-allocation of the database connection pool inside the critical microservice. Recent work [33] has proposed a Concurrency-aware system Scaling (ConScale) framework that can quickly adapt key servers' soft resource allocations after system scaling. However, ConScale is throughput

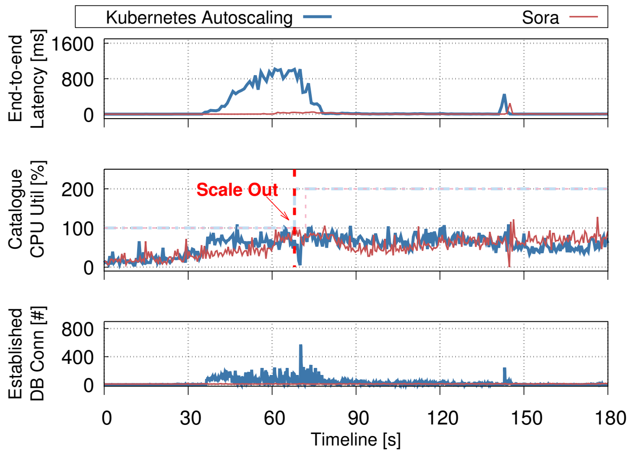


Figure 1: Large response time fluctuations of microservices due to over-allocation of soft resources (e.g., database connections) when Kubernetes Horizontal Pod Autoscaling (HPA) scales out the bottleneck service.

centric and cannot adapt the soft resource allocations for latency-sensitive microservices applications due to the lack of consideration of the runtime deadline for critical microservices along the critical paths in the system (Section 3.2).

In this paper, we propose Sora, an online soft resource adaptation management framework to quickly recommend and reconfigure soft resource allocations (e.g., server threads and database connections) for critical microservices to mitigate large response time variations. Sora leverages online fine-grained monitoring metrics (e.g., throughput, response time, and concurrency) to capture the runtime state of each component microservice and then integrates the runtime propagated deadline of critical microservices along the critical paths in the system for the prediction of rational concurrency settings.

Our Sora approach is based on two key observations. First, concurrency settings are controlled by soft resource allocations, which greatly impact the effective use of underlying hardware resources. For example, a conservative allocation (e.g., too small thread pool) may choke concurrent request processing that creates long request queues (thus longer delay), while a liberal allocation often wastes hardware resources such as CPU and memory. Second, concurrency settings have a large impact on the response time distribution of each runtime microservice; thus, the optimal setting is sensitive to the runtime deadline of critical microservices along the critical path. Figure 1 shows the need for Sora, which is applied to the runtime adaptation of a database connection pool size of the critical microservice `Catalogue` when scaling the *Sock Shop* [6] microservices benchmark application to achieve both good performance and high efficiency.

The first contribution of the paper is an empirical demonstration (based on two representative microservice benchmarks, *Sock Shop* [6] and *Social Network* [16]) that optimal concurrency settings can shift significantly under varied deadline requirements and system runtime conditions. For example, we show that a sub-optimal allocation of threads under the same hardware provisioning (e.g., CPU limit) could become the optimal allocation once we change the response time deadline from 250ms to 150ms (see Figures 3(a) and 3(b)). We also observed that, given the same response time deadline, the optimal thread pool size for a microservice instance has a non-linear increase from 10 to 30 after the CPU limit scales up from 2-core to 4-core (see Figures 3(a) and 3(c)).

The second contribution is our novel Scatter-Concurrency-Goodput (SCG) model, which integrates response time deadline into the runtime concurrency adaptation management when scaling microservices in clouds. Our model takes fine-grained runtime monitoring metrics (e.g., throughput, response time, and concurrency) as input and makes dynamic concurrency adaptation decisions based on a propagated deadline of the critical microservice, which can capture each subtle change of system runtime conditions (Section 3).

The third contribution is the design and implementation of the Sora framework, which exploits our latency-sensitive SCG model to coordinate the hardware and soft resources provisioning in microservices scaling management (Section 4). We conduct extensive experiments using six real-world bursty workload traces [17] (see Table 2), and our experimental results demonstrate that Sora can effectively alleviate large response time fluctuations and reduce the 99th percentile latencies by up to 2.5× compared to state-of-the-art hardware-only scaling strategy FIRM [47], and 1.5× to the state-of-the-art concurrency-aware system scaling strategy ConScale [33].

The rest of the paper is organized as follows. Section 2 presents experimental evidence that sub-optimal soft resource allocation leads to significant performance loss. Section 3 introduces our Scatter-Concurrency-Goodput (SCG) model. Section 4 illustrates the design and implementation of our framework Sora. Section 5 shows the experimental results under six real-world workload traces. Section 6 discusses the scalability and applicability of Sora. Section 7 summarizes the related work, and Section 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Soft Resources in Microservices

Hardware resources such as CPU, memory, and network are well-defined components in the performance evaluation of computer systems. We use the term *soft resources* to refer to the system software components such as threads and TCP connections that utilize hardware resources. For example, threads use CPU and memory, and TCP connections multiplex network I/O. In general, soft resources are key system components that control the concurrency level of a server and facilitate the sharing of hardware resources. Previous studies [37, 38, 65] have demonstrated that optimal soft resource allocation (e.g., thread pool and connection pool) plays an important role in the scalability of traditional monolithic n-tier web-facing applications in clouds because either under- or over-allocation could cause inefficient use of underlying hardware resources. Compared to traditional monolithic n-tier architecture, finding the optimal soft resource allocation for microservices-based applications is a greater challenge due to the following three reasons:

Finer-grained resource management. Unlike traditional monolithic n-tier architectures that deploy servers on physical machines or VMs, microservices adopt a more lightweight and more agile container-based virtualization technique such as Docker [14], OpenVZ [62], and Linux Containers [12]. In this case, container-based microservices can support even finer-grained resource management than VM-based applications. For example, the Kubernetes container-orchestration system provides CPU quotas level resource allocation among microservices [30]. Such finer-grained hardware resource management makes precise matching of hardware and soft resource allocations a great challenge.

Heterogeneous service implementation. Microservices-based applications are typically integrated with component services implemented in different programming languages and platforms. For example, the Cart service in the Sock Shop application [6] is built upon SpringBoot [57], which adopts an embedded thread pool to manage connections. On the other hand, the Catalogue service is written in Golang [19], which delegates connection management to asynchronous goroutines (Figure 2). The heterogeneity of component services in microservices-based applications complicates the overall optimal soft resource allocations since each component service may have its own unique characteristics and demand for optimal soft resource allocation.

Complex inter-service dependency. Compared to traditional monolithic n-tier architectures, microservices-based applications generally have a more complex inter-service dependency [36]. This is because microservices decompose the application into small, singular, and discrete services that implement business logic, resulting in a significantly large number of microservices components. For example, JD.com, China's largest e-commerce site, runs approximately 34,000 microservices on a 500,000-container cluster [32]. Each microservice typically interacts with hundreds or even thousands of other microservices, sometimes forming a long invocation chain. Such complex inter-service dependency makes it difficult to identify critical component services and conduct corresponding soft resource allocation tuning.

2.2 Experimental Setup

We adopt two representative open-sourced microservice benchmarks: **Sock Shop** [6] with 11 component microservices and **Social Network** with 36 microservices from DeathStarBench Suite [16] (see Figure 2). Sock Shop is an e-commerce website that allows customers to navigate and purchase different types of socks. Social Network is a broadcast-style social network website that allows users to publish and read social media posts. To illustrate the complexity of optimal soft resource allocations for heterogeneous component services, we evaluate three representative soft resources from the two benchmark applications – the thread pool in a SpringBoot-based microservice (i.e., Cart), the database connection pool in a Golang-based microservice (i.e., Catalogue), and the request connection pool in an Apache Thrift-based microservice (i.e., Home-Timeline). To simulate normal user access to the applications, we use the classic RUBBoS workload generator [11] to send HTTP requests. The request rate follows a Poisson distribution with the mean determined by the number of simulated users.

We run experiments in our private VMware ESXi cluster [63]. Our cluster consists of 6 bare metal servers equipped with two Intel Xeon E5-2603v3 processors (6 cores each @ 1.6GHz) and 16GB of RAM. We deployed 18 VMs in the cluster, and each VM was configured with 4 vCPUs, 4GB RAM, and 30GB disk space. Furthermore, we set up a Kubernetes cluster for container orchestration and management. We run each microservice in one container and further distribute the containers on the cluster. Docker (version 19.03.13) was used as the container runtime engine, and the Flannel network was used for Kubernetes pod networking. In this paper, we use the terms "pod" and "container" interchangeably since we use the most common one-container-per-pod model [49].

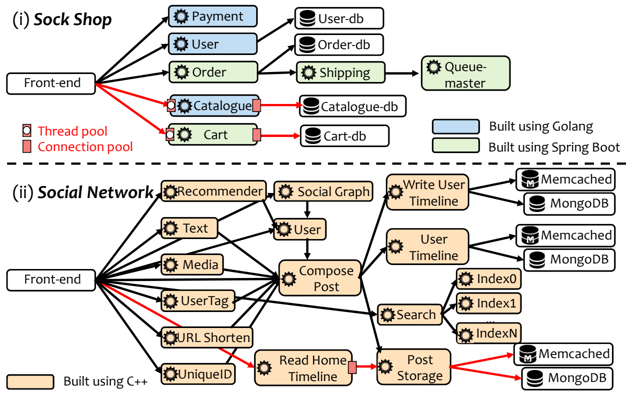


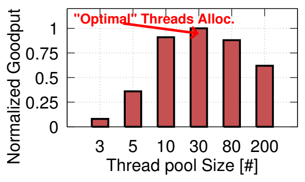
Figure 2: Architecture of *Sock Shop* and *Social Network* microservice benchmark applications.

2.3 Shifting of Optimal Soft Resource Allocation during Runtime

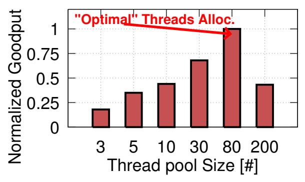
Industry practitioners usually conduct offline parameter tuning [65] or sandboxing [25] to identify the appropriate soft resource allocations (i.e., concurrency settings) in microservices to improve resource efficiency and meet their performance expectations. Web-based e-commerce applications are typically latency-sensitive, whereas Service Level Agreements (SLAs) are commonly used to specify the desired response time of user requests to avoid potential revenue loss. In this paper, we use a simplified SLA model to illustrate the revenue tradeoffs between throughput and response time and further evaluate the appropriate concurrency settings. Our results suggest that static soft resource allocations are not efficacious forever in the production phase. Fast and accurate runtime optimal soft resource adaptation is critical to realizing stable system performance.

For our simplified SLA model, we set a single threshold for the end-to-end response time of requests (e.g., 250ms). We only count requests with a response time equal to or below the threshold, defined as *goodput*. On the contrary, the sum of requests with their response time higher than the threshold is defined as *badput*. The sum of goodput and badput amounts is the traditional definition of throughput. The goodput model provides a better quantitative performance evaluation of latency-sensitive web applications than the throughput model since it considers the potential revenue loss due to long response time. Then we rely on the goodput model to tune the soft resource allocations in the benchmark applications and observe factors that may lead to the shifting of "optimal" concurrency settings during system runtime.

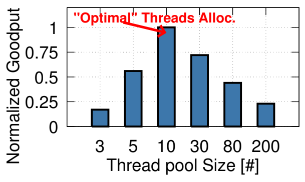
Response Time Threshold Changing. We initially set a 250ms response time threshold for requests accessing the Cart service from



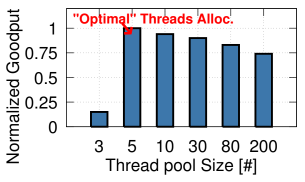
(a) **4-core** Cart service with a **250ms threshold**. 30 threads allocation achieves the highest goodput.



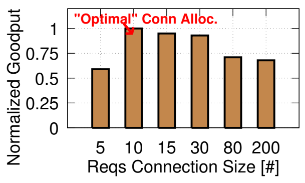
(b) **4-core** Cart service with a **150ms threshold**. 80 threads allocation achieves the highest goodput.



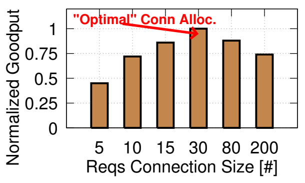
(c) **2-core** Cart service with a **250ms threshold**. 10 threads allocation achieves the highest goodput.



(d) **2-core** Cart service with a **350ms threshold**. 5 threads allocation achieves the highest goodput.



(e) Post Storage service serving **light** requests. 10 connections allocation achieves the highest goodput.

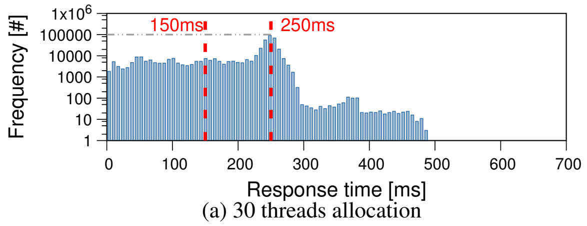


(f) Post Storage service serving **heavy** requests. 30 connections allocation achieves the highest goodput.

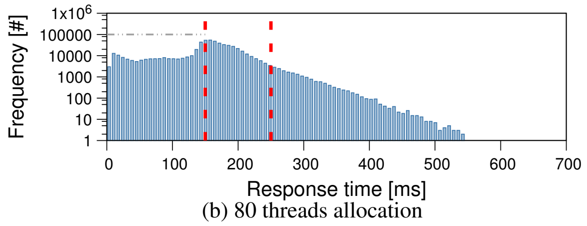
Figure 3: "Optimal" soft resource allocation shifts for *Cart* and *Post Storage* as response time threshold, hardware provisioning, or system state changes.

Sock Shop. Figure 3(a) shows that 30 threads allocation can achieve the highest goodput in a 3-minute experiment since too few threads cannot fully utilize the hardware resources (e.g., the 4-core CPU or 4 vCPU). In the meantime, too many threads result in performance degradation due to non-trivial multithreading overhead [65]. However, in Figure 3(b), the "optimal" threads setting shifts to 80 when we reset the response time threshold to 150ms. Such a shifting of "optimal" threads setting is also observed in the 2-core Cart service once we change the response time threshold from 250ms to 350ms, shown in Figures 3(c) and 3(d).

The response time threshold affects the goodput measurement because different threads allocation may lead to different response time distributions. Figures 4(a) and 4(b) show the response time distribution graphs when the 4-core Cart service was allocated 30 and 80 threads, respectively. By comparing the two distributions, the 80-thread case clearly achieves a higher goodput than the 30-thread with the response time threshold of 150ms. However, the performance order is reversed once we reset the response time threshold to 250ms since more requests fall within 250ms in the 30-thread case than that in the 80-thread case. Such experimental results have two implications: (1) the response time threshold has a large impact on the optimal concurrency setting based on goodput; (2) adapting concurrency settings according to changing



(a) 30 threads allocation



(b) 80 threads allocation

Figure 4: Semi-log graphs of request response time distribution of the 4-core Cart service configured with different threads allocations. The 80-thread case outperforms the 30-thread case with a response time threshold (RTT) of 150ms since the peak dominates the total amounts of requests, but the performance order reverses with RTT 250ms.

response time thresholds for microservices in the production phase is necessary.

Hardware Resource Scaling. The vertical scaling (e.g., adding or removing # of CPU cores) would affect the optimal concurrency setting of a microservice instance. Figures 3(a) and 3(c) show that the "optimal" thread pool allocation to reach the highest goodput (with 250ms response time threshold) shifts from 10 to 30 when the CPU limit for the Cart service scales up from 2-core to 4-core. This is because the original optimal concurrency setting (i.e., 10 server threads) becomes under-allocation and cannot fully utilize 4 CPU cores after the system scaling. We have observed consistent experimental results about the shift of optimal database connection pool allocation in the Catalogue service as the CPU limit scales up.

System State Drifting. The system state of the backend services could drift over runtime due to continuous dataset updates or software upgrades, which leads to variations of the service time of the involved service accordingly. Here we evaluate the Post Storage service from Social Network and manually control the number of posts accessed by the users to evaluate the impact of requests with a light (retrieve 2 posts) and heavy (retrieve 10 posts) computation on the optimal soft resource allocation. The computation for each user request is proportional to the number of accessed posts. Figures 3(e) and 3(f) show the optimal connections to the Post Storage service (from the upstream Home Timeline service) shift from 10 to 30 once the same type of requests changes from light to heavy due to the state drifting of the dataset.

The above empirical observations demonstrate that different soft resource allocations incur large performance variations. The optimal soft resource allocation is always changing during system runtime, depending on factors such as response time threshold selection and runtime system conditions. Due to the naturally bursty workload and the frequent hardware resource scaling in cloud environments, service providers need an online model that can quickly

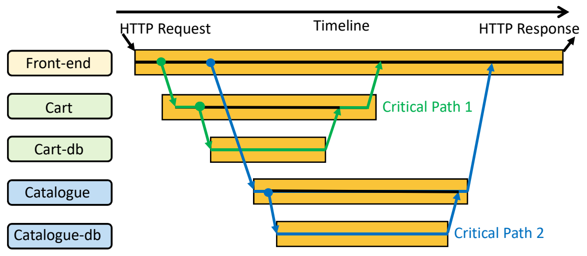


Figure 5: An illustration of the execution path of a *Catalogue* request in the Sock Shop benchmark.

and accurately provide appropriate soft resource allocations during runtime.

3 SCATTER-CONCURRENCY-GOODPUT MODEL

This section introduces an online Scatter-Concurrency-Goodput (SCG) Model for optimal concurrency setting estimation for critical microservice instances, which can resolve the limitations of existing online approaches for concurrency adaptation (Section 3.1). Our model correlates fine-grained application-level metrics (i.e., request processing concurrency and goodput) within a short time window (e.g., 3 minutes) and quickly recommends a concurrency setting that can achieve the highest goodput (Section 3.2). We further provide a sensitivity analysis of our SCG model (Section 3.3). Our model aims to guarantee the response time requirements of critical microservices with concurrency settings adaptation.

3.1 Limitations of Existing Online Models

Existing online models identify optimal concurrency settings by building and revising the performance model during runtime [25, 33, 58] or applying a step-by-step heuristic approach such as Bayesian optimization [10, 55, 75]. One big limitation is that these models are throughput oriented. For example, ConScale [33] adopts an online model to estimate the optimal resource allocations for each service based on the correlation between the real-time fine-grained throughput and concurrency of component servers within a 3-minute time window. However, such a latency-agnostic throughput-based model may not satisfy the SLOs of the microservices-based applications. Nevertheless, fast adapting concurrency in microservices to avoid SLO violations suffers from two challenges.

First, response time variation can be significantly amplified due to the long invocation chain of service dependencies in microservices-based applications. Compared to the traditional monolithic architecture, the granularity of each microservice is much smaller, and the depth of the call invocation chain becomes much deeper than that in monolithic systems, leading to complex inter-service dependency (call graphs) and further incurring the latency long-tail phenomenon [36, 50, 58].

Second, microservice call graphs can be highly dynamic during runtime [36, 47]. Specifically, microservices-based applications can present significant topological differences during runtime (even for the same online service), leading to variances in **critical paths** [1].

[1]A critical path to a call graph (triggered by a user request) is defined as the path of maximal duration that starts with the user request and ends with the final response.

For example, either Cart or Catalogue can become the critical path in the execution of a *Catalogue* request (see Figure 5), depending on the underlying resource contention and external user inputs. The dynamic behavior of critical paths would amplify the end-to-end response time variations, which impacts the goodput measurement and decreases the accuracy of model estimation.

3.2 Model Description

Our Scatter-Concurrency Goodput (SCG) model addresses the limitations of classic online throughput-based models in two ways. First, the SCG model uses goodput to improve the model sensitivity. This is because the goodput measurement requires a response time threshold, which takes response time into account. For example, Figure 6(a) (in phase ④) characterizes the theoretical relationship (known as the *main sequence curve*) between the goodput and the concurrency of a microservice instance. The server goodput increases almost linearly as the concurrency increases until it reaches the maximum. As the concurrency continues to increase, the server goodput starts to decrease because the increasing response time exceeds the pre-defined threshold, and the requests with long response times do not count into the goodput measurement. Therefore, the maximum goodput is highly related to the response time threshold.

Second, the SCG model can handle large variations of metrics measurement by filtering out partial "noisy" requests with long response times. Due to the complex system dynamics, large variations of the classic concurrency-throughput pairs adopted by online throughput models (without filtering out "noisy" requests) make the main sequence curve extraction extremely difficult from the scatter graph [33]. On the other hand, the goodput model can easily localize the knee point from the scatter graph after filtering the requests with bounded response time (see Figure 6(b) in phase ④).

Our SCG model identifies the optimal concurrency setting based on a statistical analysis of each microservice's real-time goodput and concurrency. Figure 6 shows an overview of the SCG model and illustrates the four major phases in the online optimal concurrency estimation workflow.

Critical Service Localization Phase aims to quickly and accurately identify the critical path in the request call graph and further localize the critical service for concurrency adaptation. The critical path should be the path with the longest processing time in the request execution graph, which occupies a significant portion of the end-to-end response time. The *critical service* refers to the bottlenecked microservice on the identified critical path. Inspired by FIRM [47], we adopt a two-step method to localize the critical service. First, we evaluate the resource utilization of each microservice. The high resource utilization indicates a candidate critical microservice reaches its capacity, which possibly leads to request congestion. Second, we profile the processing time of the i-th microservice PT_{s_i}, and the end-to-end response time of the critical path RT_{CP}, based on the arrival and departure timestamps of each request. We further calculate the Pearson correlation coefficient [9] of each microservice's processing time and the end-to-end response time of the critical path (i.e., $PCC(PT_{s_i}, RT_{CP})$). We consider the microservice that has the largest value as a candidate critical service since a large value indicates that the corresponding microservice contributes more to the end-to-end latency variation. In fact, the

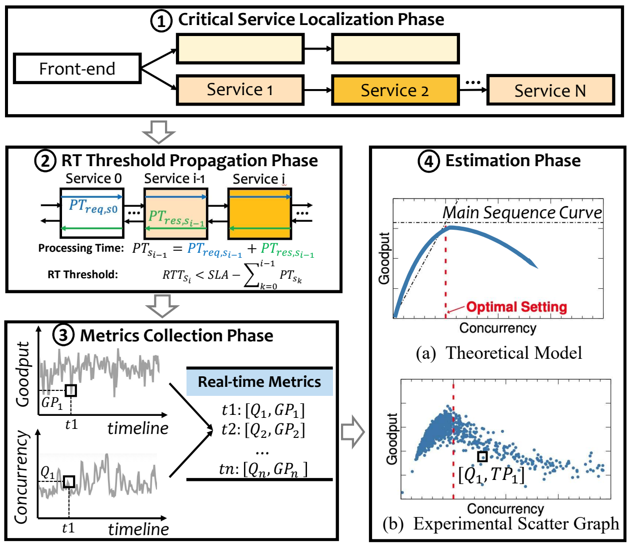


Figure 6: Overview of SCG Model and its workflow for optimal concurrency estimation in microservices.

critical services recommended by both steps overlap most of the time in our experiments.

RT Threshold Propagation Phase determines the response time threshold (i.e., deadline) for critical service goodput measurement. We apply a deadline propagation algorithm [50] to the critical path. The key idea of deadline propagation is to allow local services to perceive the global deadline information within a critical path and leverage the information to adjust soft resources to reduce latency variation. Since requests flow among microservices by following parent-child relationship chains and each microservice would call its child downstream microservices when handling a parent call from its upstream microservice, we use i to denote the depth of service and consider the front-end service as service 0. For any service s_i in the path, we regard the parent service s_{i-1} as the upstream service and the child service s_{i+1} as the downstream service. Suppose the network latency between services is negligible and the critical service is s_i, the end-to-end response time RT is composed of the request processing time of all upstream services $\Sigma_{k=0}^{i-1} PT_{\boldsymbol{req},s_k}$, the response time of current service RT_{s_i}, and the response processing time of upstream services $\Sigma_{k=0}^{i-1} PT_{\boldsymbol{res},s_k}$. Then according to the SLA requirements, we have

$$\Sigma_{k=0}^{i-1} PT_{\boldsymbol{req},s_k} + RT_{s_i} + \Sigma_{k=0}^{i-1} PT_{\boldsymbol{res},s_k} \leq SLA \tag{1}$$

Considering that the sum of request processing time $PT_{\boldsymbol{req},s_i}$ and response processing time $PT_{\boldsymbol{res},s_i}$ is the total processing time of the i-th service PT_{s_i}, we further simplify the equation as

$$\Sigma_{k=0}^{i-1} PT_{s_k} + RTs_i \leq SLA \tag{2}$$

The response time threshold (RTT) for service i should be

$$RTT_{s_i} \leq SLA - \Sigma_{k=0}^{i-1} PT_{s_k} \tag{3}$$

We notice that the response time threshold of the critical service i is only determined by the processing time of upstream services. We record the timestamps of each message (including requests and responses) that arrives/leaves each service, so the sum of the processing time of all upstream services can be calculated. Then the response time threshold of critical service will not be affected by the inter-dependency of upstream services. Take the Cart service as an example (see critical path 1 in Figure 5), suppose the SLA requirement of the *Cart* request is 150ms, and we identify the Cart service as the critical service. We then measure the processing time of the front-end service (i.e., upstream of Cart) as 10ms, so the response time threshold of the Cart service should be 140ms.

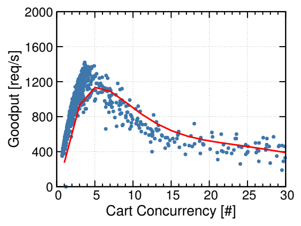
Metrics Collection Phase calculates goodput through 1) the fine-grained measured throughput (e.g., at 100ms granularity) and 2) the deadline information of the critical service extracted in the previous phase. We collect a series of pairs $< Q_n, GP_n >$ sampled at 100ms granularity within a short period (e.g., 3 mins) to generate the scatter graph, shown in Figure 6(b). For a specific server concurrency Q_n, we calculate the average goodput $\overline{GP_n}$. After that, a series of pairs $< \overline{Q_n}, \overline{GP_n} >$ are prepared for the next phase. We note that we do not need to specifically configure the range of goodput and concurrency since the real-time concurrency of each microservice varies significantly due to dynamic workload. Due to the naturally bursty workload, the shape of the detailed goodput-concurrency curve of the critical service would naturally appear within a few seconds. We notice that too-conservative concurrency settings may affect knee point detection since insufficient concurrency cannot fully utilize the hardware resources (e.g., CPU) to reach the maximum goodput and further blurs the knee point. Hence, we gradually increase the allocation to find a new optimal value. Some advanced exploration policies will be explored in our future work.

Estimation Phase finds the optimal concurrency setting (knee point) from a discrete data set (e.g., a series of data pairs $< \overline{Q_n}, \overline{GP_n} >$ from the metric collection phase). We consider the knee point of the Main Sequence Curve (see Figure 6(a)) to be the optimal concurrency of the corresponding critical service, which can achieve the highest goodput within the requested deadline (from the RT threshold propagation phase). For example, Figure 7[2] shows the correlations of Cart concurrency and goodput measured at the 100ms time granularity over the same 3-minute runtime under a bursty workload with a 5ms and a 50ms response time threshold, respectively. We observe that a high response time threshold (i.e., 50ms) leads to a different knee point from that with a low response time threshold (e.g., 5ms) since the goodput measurement is highly sensitive to the response time threshold selection.

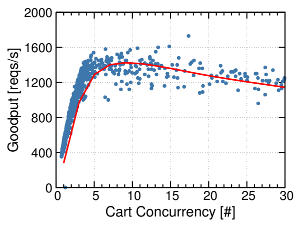
3.3 Sensitivity Analysis of SCG Model

To provide a fast and accurate optimal concurrency estimation, we apply a simple statistical approach, Kneedle [53], to detect the knee point of the correlation between concurrency and goodput. The knee point is defined as the local maximum of curvature, which may occur when a curve becomes more "flat". In practice, we need to tune the *polynomial_degree* for each service to provide an accurate estimation. Kneedle uses a smoothing spline to preserve the shape of the original data set, so the degree of polynomial fit will affect the accuracy of knee point estimation. A too-low polynomial degree cannot provide a valid knee point while a too-high polynomial

[2]We use Gnuplot [60] smooth function for data smoothing with the Bezier or the cubic-splines curves



(a) Correlation between Cart concurrency and goodput with 5ms threshold.



(b) Correlations between Cart concurrency and goodput with 50ms threshold.

Figure 7: The correlations between Cart concurrency and goodput were measured at 100ms granularity during a 3-minute experiment with different response time thresholds. The red line is the trend line for each scatter graph.

Table 1: Optimal concurrency estimation accuracy of SCG model with various sampling intervals for Cart, Catalogue, and Post Storage.

Sampling Interval		10ms	20ms	50ms	100ms	200ms	500ms
MAPE [%]	Cart	16.67	15.00	13.33	5.83	10.83	15.00
	Catalogue	14.67	10.67	7.99	5.33	13.33	17.33
	Post Storage	19.48	18.18	17.69	12.04	13.07	15.38

degree (e.g., 20) would lead to overfitting (i.e., fitting the data noise), thus degrading the quality of estimation. We adopt an incremental tuning strategy to quickly identify the minimum polynomial degree to generate an estimation that matches the profiling data. In our case, a polynomial degree ranging from 5 to 8 can fit our profiling data (e.g., 1-minute data) quite well with the sub-second time cost.

Besides, goodput and concurrency sampling interval (e.g., 100ms) is another important parameter affecting the model estimation accuracy. Too long a sampling interval cannot capture the transient variation of concurrency while too short a sampling interval may incur large variations for goodput and concurrency measurements that degrade the knee point estimation. In addition, extensive data points due to short sampling intervals may lead to additional CPU and memory overhead for online estimation. We evaluate the estimation accuracy of our SCG model with various sampling intervals on three services and compare their Mean Absolute Percentage Errors (MAPE [2]), shown in Table 1. We observe that 100ms can achieve the best estimation accuracy with the minimum MAPE for all three services. An automatic way to choose a proper time interval that minimizes the MAPE for all types of microservices is our future research.

4 SORA FRAMEWORK

In this section, we describe the design and implementation of our framework Sora, which integrates our SCG model to work with a hardware-only autoscaler to mitigate the large response time fluctuations when handling bursty workloads (see Figure 8). Sora first detects critical services based on runtime metrics (e.g., CPU utilization, request/response timestamps) collected by **Monitoring Module** and evokes hardware-only autoscaler inside **Reallocation Module** to arrange hardware resource scaling. Then the autoscaler signals the **Concurrency Estimator** to query the optimal soft

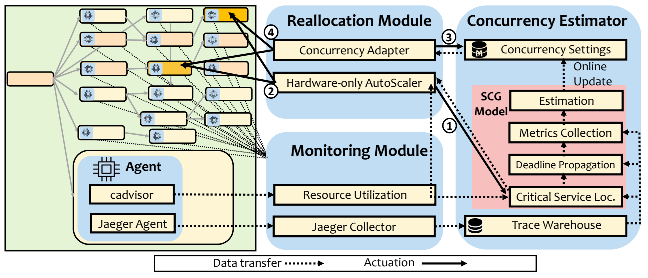


Figure 8: Sora framework for runtime adapting of optimal concurrency settings for critical microservices.

resources allocation for corresponding services and further triggers soft resource adaptation. Meanwhile, the Concurrency Estimator updates the optimal soft resources settings during runtime by extracting the fine-grained application-level metrics for the SCG model (Section 3) to conduct online estimation. We further describe the design of each module in the following text.

4.1 Module Design

Monitoring Module collects both system-level and performance metrics through a monitoring agent installed in each microservice instance. The system-level metrics include hardware resource utilizations (e.g., CPU, network I/O, memory) for the hardware-only autoscaler inside Reallocation Module since many cloud and service providers (e.g., Amazon EC2 AutoScaling and Kubernetes HPA/VPA) use these metrics as the scaling indicator [48]. Meanwhile, the Monitoring Module also adopts distributed tracing tools to record individual requests' arrival and departure timestamps within each microservice at millisecond granularity. We implement an OpenTracing-compliant tracing module inside each microservice inspired by Jaeger [24] and Zipkin [77]. We stored the request traces in Trace Warehouse for fine-grained performance metrics extraction.

Concurrency Estimator is to update optimal soft resource allocation based on our SCG model estimation during runtime. It continuously pulls trace data from the Trace Warehouse for critical service localization and RT threshold propagation phases inside of SCG model asynchronously. Processing these request timestamps can further generate fine-grained concurrency and goodput within the time window for model estimation. Such a time window should keep a balance between long enough to have sufficient metrics for generating the entire trend line and short enough to keep the model agile to the changes in workload characteristics and system conditions. Based on the 100ms sampling interval and control period of hardware-only autoscaler (e.g., default 15s in Kubernetes HPA [7]), we configure a 60s time window setting, which can accumulate 600 data points for knee point estimation.

Reallocation Module includes a hardware-only autoscaler to manage system hardware resources and the Concurrency Adapter to reallocate the soft resources recommended by the Concurrency Estimator. Once the hardware-only autoscaler is triggered, it signals the SCG Model to identify the critical services and respond to the autoscaler for scheduling the hardware resource scaling. The Concurrency Adapter then queries the concurrency settings of critical services from the Concurrency Estimator and applies soft resource reallocation after hardware resource scaling. The independent design of applying hardware scaling and concurrency adaptation is to easily integrate our SCG model to the state-of-the-art hardware-only scaling solutions (e.g., Kubernetes HPA/VPA [7] and FIRM [47]). A unified controller can potentially be an ideal solution for this joint optimization problem, which is subject to our future work.

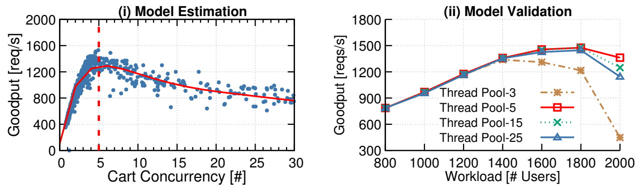
4.2 Implementation Details

Request Tracing Management. Our SCG model relies on distributed tracing to collect detailed request information to generate fine-grained performance metrics to update optimal soft resource allocation. However, efficient management of real-time trace data in a large-scale microservice system is a great challenge [21, 28, 74]. To mitigate this problem, we first use a graph database, Neo4j [4], to efficiently store and query the complex invocations of services. Furthermore, we prepare a separate lightweight database (e.g., MongoDB [3]) for each microservice to store the request/response timestamps of the current service, which removes the overhead of heavy filtering and aggregation that occurs in centralized storage. Meanwhile, we isolate the resource (e.g., CPU) for the monitoring agent with microservices in each VM to avoid interference.

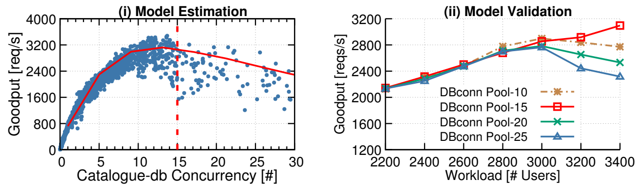
Runtime Soft Resource Reallocation. Soft resources such as threads and network connections can be dynamically reallocated during runtime via flexible APIs provided by open-sourced software and third-party libraries. For example, we can adjust the thread pool size for the SpringBoot-based Cart service through remote JMX access via *Jolokia* [26] and manage the database connection pool size in the Golang-based Catalogue service through a manually extended service via calling APIs in Golang package "database/sql". On the other hand, some applications tend to provide generic interfaces for dynamically configuring service parameters. For example, the Apache Thrift-based Social Network employs a ClientPool class to configure the settings of connections among services, such as the number of connections and timeout. Other internal (or deep) soft resources such as locks are more application-specific, and exposing these internal soft resources to the autoscaler requires extra effort by service developers.

5 EXPERIMENTAL EVALUATION

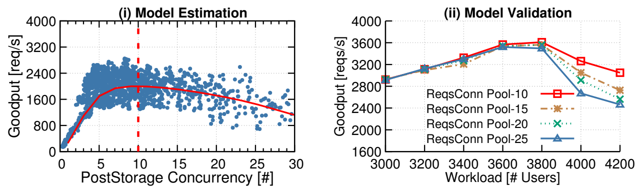
In this section, we first validate the generality of our SCG model on different types of soft resources (e.g., threads and network connections) using two representative benchmark applications (i.e., Sock Shop and Social Network) (Section 5.1). We then evaluate the effectiveness of Sora in assisting the hardware-only autoscaler in stabilizing performance fluctuations under six real-world bursty workload scenarios [17] (see Table 2). Concretely, we present that Sora integrated a state-of-the-art hardware-only autoscaler FIRM [47], which can effectively mitigate the response time fluctuations due to runtime system condition variations (Section 5.2 and 5.3). We also compare Sora with the state-of-the-art concurrency-aware concurrency adaptation framework ConScale [33], confirming that Sora can achieve higher goodput than the classic throughput-based model scaling management (Section 5.2). In this section, we use the same experimental setup in Section 2.2.



(a) SCG model recommends 5 threads for Cart can achieve the highest goodput with 10ms threshold.



(b) SCG model recommends 15 database connections for Catalogue can achieve the highest goodput with 10ms threshold.



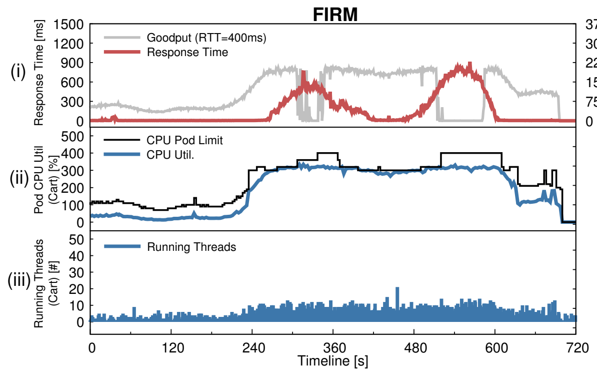
(c) SCG model recommends 10 request connections to Post Storage can achieve the highest goodput with 15ms threshold.

Figure 9: Validating our SCG model estimation of threads in Cart (a) and connections in Catalogue (b) and Home Timeline (c) using realistic system configuration. These results demonstrate that our model estimation outperforms the other three adjacent allocations in all three services.

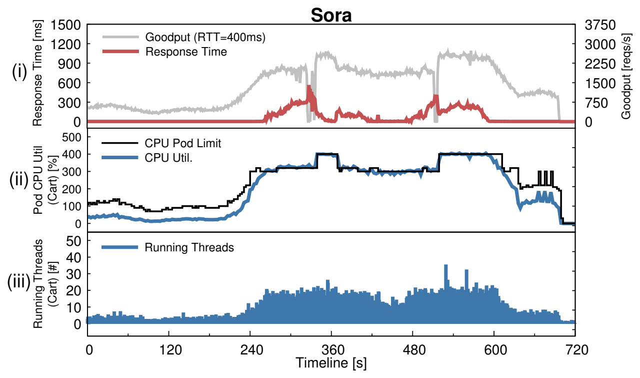
5.1 Model Validation for Different Soft Resources Estimation

We show three case studies on the effectiveness of our SCG model in providing accurate optimal concurrency estimation for different critical soft resources (e.g., threads and connections). In this set of experiments, we first apply our SCG model to different services (left column of Figure 9) to estimate the optimal concurrency settings for different critical soft resources. We then validate the soft resource allocations recommended by SCG through extensive experiments (right column of Figure 9).

Case 1: Threads in Cart. Our first case study is to estimate the thread pool size for the SpringBoot-based Cart service from Sock Shop. To correctly obtain the optimal allocation of thread pool in Cart service using our SCG model, we correlate the runtime goodput and request processing concurrency in Cart to generate a scatter graph for optimal concurrency estimation as shown in Figure 9(a)(i). The figure shows that the SCG model recommends 5 threads as the optimal thread allocation in Cart service, which is sufficient to fully utilize CPU and guarantee SLAs. We validate the Cart thread pool size recommendation in Figure 9(a)(ii). In this evaluation, we gradually increase the Cart thread pool size from 3 to 25. The figure shows that the *Thread Pool-5* achieves the highest



(a) The system response time spikes appear during 269s~412s, 480s~610s under the "Steep Tri Phase" workload. The goodput drops during 307s~345s, 512s~583s. FIRM manages the CPU limit for the Cart service. The Pod CPU resources are under-utilized after Cart scales up to 4-core due to the under-allocation of server threads.



(b) Relatively stable system response time under the same workload trace in (a) with Sora. Response time spikes appear during 269s~412s, 480s~590s. The goodput only drops at 333s and 513s. The Pod CPU resources can be fully utilized after Cart scales up to 4-core due to optimal thread pool reallocation conducted by Sora.

Figure 10: Performance comparison between FIRM and Sora under the same "Steep Tri Phase" workload. Figure 10(a) is for FIRM and Figure 10(b) is for Sora. Our framework Sora can help FIRM stabilize response time fluctuation by re-adapting thread pool allocation to match the bursty workload.

goodput, suggesting that either under- or over-allocation of the thread pool could lead to inefficient usage of underlying hardware resources.

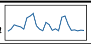
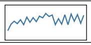
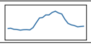
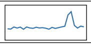
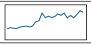
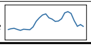
Case 2: Database Connections in Catalogue. Our second case study is to estimate the database connection pool size for the Golang-based Catalogue service from Sock Shop. In Figure 9(b)(i), our SCG model suggests that 15 database connections in the Catalogue service can achieve the highest goodput while the response time threshold for goodput is 10ms. Such a recommendation is also validated through our extensive evaluation in Figure 9(b)(ii).

Case 3: Request Connections in Post Storage. Our third case study is to estimate the ClientPool (request connections) size for the Apache Thrift-based Post Storage service from Social Network. Figure 9(c)(i) shows the correlation of goodput and concurrency in Post Storage during a 3-minute experimental period. Our SCG model recommends 10 request connections in Post Storage as the optimal concurrency setting with a 15ms response time threshold. Our validation in Figure 9(c)(ii) confirms that such request connection pool allocation to Post Storage is indeed the optimal concurrency setting compared to other candidates.

5.2 Mitigating Response Time Fluctuations in Autoscaling

We validate our design by deploying Sora, FIRM, and ConScale in our private testbed (see Section 2.2). We implement a prototype of Sora that uses FIRM as the underlying hardware-only management framework. FIRM [47] provides an RL-based fine-grained hardware resource management for microservices. ConScale [33] is the state-of-the-art online concurrency adaptation framework, which correlates the runtime service throughput and concurrency for fast concurrency adaptation with autoscaling to stabilize the system's response time during runtime. In this evaluation, we use the Cart service from the benchmark application Sock Shop, and the maximum number of concurrent users for the Cart service is

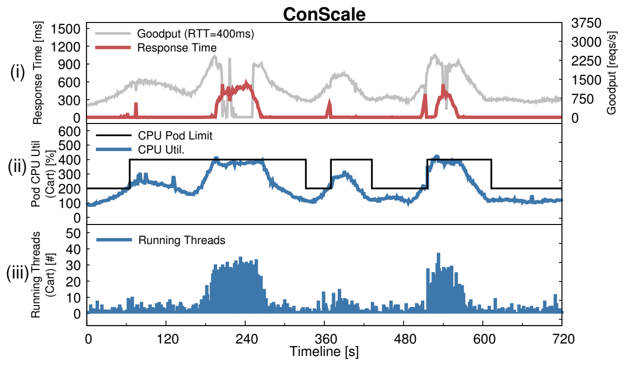
Table 2: Tail response time (i.e., 95th and 99th percentile) and average goodput comparison between FIRM and Sora under six real-world bursty workload traces. The results show Sora helps FIRM improve the goodput and reduce the tail response time.

Workload Trace		95th Percentile Response Time [ms]	99th Percentile Response Time [ms]	Goodput-400ms [reqs/s]
		FIRM / **Sora**	FIRM / **Sora**	FIRM / **Sora**
Large Variation		501 / **230**	592 / **278**	913 / **1172**
Quick Varying		500 / **247**	553 / **314**	1222 / **1518**
Slowly Varying		663 / **303**	749 / **400**	589 / **730**
Big Spike		535 / **218**	642 / **358**	618 / **659**
Dual Phase		551 / **319**	633 / **354**	705 / **870**
Steep Tri Phase		624 / **286**	687 / **321**	819 / **1012**

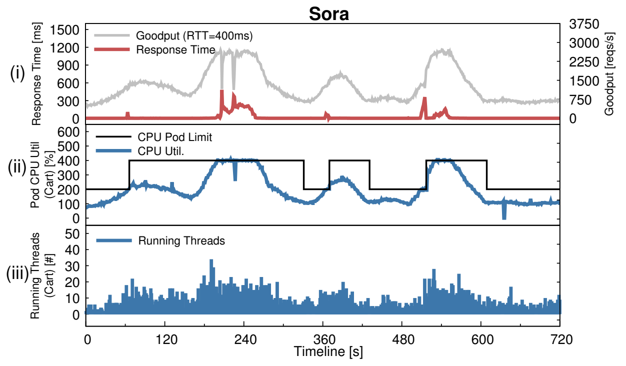
3500. The duration of each workload trace is 12 minutes. Our experimental results demonstrate that scaling microservices in clouds to achieve good performance and high efficiency requires careful runtime concurrency adaptation.

FIRM vs. Sora. Figure 10 shows the performance (e.g., goodput and response time) comparison between FIRM and Sora under the same "Steep Tri Phase" workload trace for the Cart service. The left three figures (Figure 10(a)) show the FIRM case and the right figures (Figure 10(b)) show Sora case. In this set of experiments, we initially set the thread pool size in Cart to be 5, which is reasonable for the 2-core CPU limit scenario through pre-profiling in Section 5.1.

Sora achieves a relatively stable response time and goodput in a 12-minute experiment than that in the FIRM case (see Figure 10(a)(i) and 10(b)(i)). For example, large response time fluctuations and goodput drops in the FIRM case during the temporary overload phase (307s~345s and 512s~583s). Taking the period 512s~583s in Figure 10(a) as an example, as the workload continues to increase at 520s, we note that the number of incoming requests accumulates and significantly affects the system performance. After the Pod



(a) The system response time spikes appear during 180s∼260s, 500s∼570s under the "Large Variation" workload. ConScale adapts 40 threads when Cart scales to 4-core to maximize throughput. However, such a liberal allocation leads to goodput drops due to SLO violations.



(b) Relatively stable system response time under the same workload trace in (a) with Sora. Sora only limits 30 threads after Cart scales to 4-core based on the SCG model estimation with the latency constraint.

Figure 11: Performance comparison between ConScale and Sora under the same "Large Variation" workload. Figure 11(a) is for ConScale and Figure 11(b) is for Sora. Our framework Sora outperforms ConScale with a higher goodput since Sora adopts a latency-sensitive concurrency adaptation.

Table 3: Average goodput comparison between ConScale and Sora under six real-world bursty workload traces. The results show Sora outperforms ConScale to achieve higher goodput.

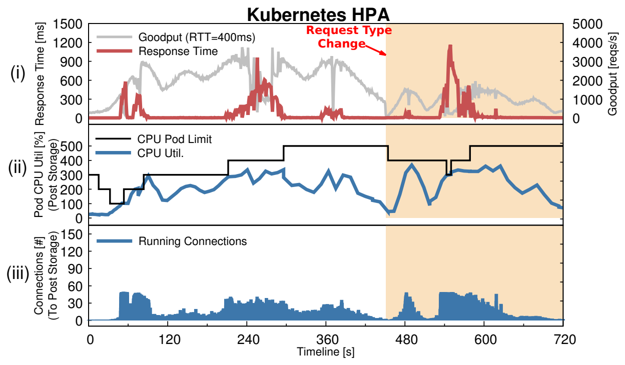
Goodput [reqs/s]		*Large Variation*	*Quick Varying*	*Slowly Varying*	*Big Spike*	*Dual Phase*	*SteepTri Phase*
SLA Threshold	*ConScale*	985	1426	657	636	1066	934
250ms	**Sora**	**1207**	**1686**	**1007**	**677**	**1251**	**1222**
SLA Threshold	*ConScale*	1122	1712	860	669	1379	1151
500ms	**Sora**	**1283**	**1886**	**1190**	**711**	**1498**	**1395**

CPU resources for the Cart instance scales up to 4-core, we found that the newly added CPU core cannot efficiently process a high volume of concurrent requests. This is because FIRM lacks the capability of adapting the thread pool accordingly after reallocating hardware resources, making the original optimal allocation of the thread pool insufficient to fully utilize the hardware resources (e.g., CPU utilization in Figure 10(a)(ii)) as we studied in Section 5.1. For example, the CPU utilization of Cart is about 310% even though the CPU limit is scaled up to 4-core, leading to the Cart CPU's low efficiency and sub-optimal system performance. On the other hand, Sora can easily cooperate with FIRM as the concurrency adapter inside Reallocation Module coordinates with the hardware-only autoscaler. It dynamically adapts the thread pool within the Cart service to a rational level along with various vCPU allocations in Figure 10(b)(ii).

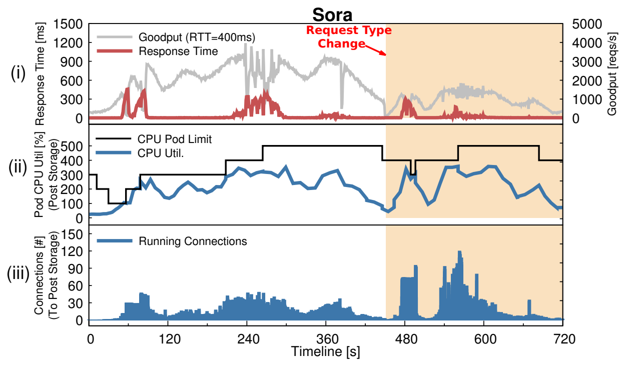
We further compare the average goodput and tail response time (i.e., 95th and 99th percentile) between the hardware-only scaling frameworks and our framework Sora under other workload traces in Table 2. Our results indicate that Sora can significantly reduce the 95th and 99th percentile response time by 2.2× on average than FIRM. In the meantime, Sora can assist hardware-only scaling frameworks in achieving a goodput improvement as the goal of our SCG model.

ConScale vs. Sora. We also validate the effectiveness of our proposed Sora framework in mitigating response time fluctuations compared with the state-of-the-art concurrency-aware system scaling framework, ConScale. We configure both ConScale and Sora to adopt a simple threshold-based hardware scaling solution (i.e., Kubernetes VPA). Table 3 shows a goodput comparison between ConScale and Sora under the same six real-world bursty workload traces, demonstrating that Sora can provide higher goodput than the ConScale framework. The goodput observed in Table 3 is higher than that in Table 2 since Kubernetes VPA allocates much more hardware resources to react to the bursty workloads. Take the "Large Variation" case as an example, Figure 11(a)(i) shows the performance comparison between ConScale and Sora under the "Large Variation" workload trace [17]. For example, Figure 11(a)(i) shows that ConScale with runtime concurrency adaptation still experiences large response time spikes during the temporary overload phase (e.g., periods 180s∼260s and 500s∼570s). This is because ConScale uses a latency-agnostic throughput-based model (i.e., Scatter-Concurrency-Throughput (SCT) model) without taking response time into account, leading to the over-allocation of the thread pool and inefficient CPU utilization as shown in Figure 11(a)(iii) and (ii), respectively. In contrast to the SCT model, our goodput-based SCG model takes the response time constraints of Cart into consideration and recommends more rational concurrency allocations (i.e., 30 threads), which can also fully utilize the CPU resources and reduce SLO violations (see Figure 11(b)).

Readers may wonder whether we can simply replace the throughput with goodput to enable ConScale for SLA-based soft resource adaptation. The answer is no because goodput calculation requires appropriate latency threshold selection during runtime, as shown in Section 2.3. In a large-scale microservices-based system, the latency requirements of critical services may change over time due to the dynamic nature of microservices. Hence, one important contribution of our SCG goodput model is to dynamically identify the critical service and its runtime latency requirements and then nicely integrate both the throughput and latency requirements into a simple model.



(a) The system response time spikes appear during 45s∼82s, 222s∼295s, 508s∼578s under the "Large Variation" workload even with a threshold-based autoscaler to scale the critical service horizontally. The static 50 connections to Post Storage would become insufficient and cause performance degradation after the system state drifting at 451s.



(b) Relatively stable system response time under the same workload trace in (a) with Sora. Sora dynamically adjusts connection pool size based on the current parallelism of Post Storage and updates the optimal connections based on SCG model estimation after system state drifting.

Figure 12: Performance comparison between Kubernetes HPA and Sora under the same "Large Variation" workload. Figure 12(a) is for Kubernetes HPA and Figure 12(b) is for Sora. Our framework Sora can dynamically adjust soft resources (e.g., request connections) to adapt to the new system state due to request type change and stabilize the response time.

5.3 Mitigating Response Time Fluctuations in System State Drifting

In this section, we validate the effectiveness of Sora in mitigating response time fluctuations when the microservices-based applications face the system state drifting (e.g., request type change). To avoid the impact of vertical scaling on optimal soft resource allocation, we deploy the Kubernetes Horizontal Pod Autoscaling (i.e., HPA) [7] in our private testbed to maintain the quality of service during runtime. The Kubernetes Autoscaling employs a rule-based scaling policy by monitoring resource utilization of microservice instances (e.g., Pod CPU utilization > 80%). We conduct evaluation experiments using the Post Storage service from the benchmark application Social Network, and we set the maximum number of concurrent users for the Read HomeTimeline service to 4500.

Figure 12 shows the performance comparison between Kubernetes HPA and Sora under the same "Large Variation" workload trace for the Read HomeTimeline service. The left three figures (Figure 12(a)) show the Kubernetes HPA case, and the right figures (Figure 12(b)) show Sora case. In this set of experiments, we initially set the request connection pool size to be 10 for each Post Storage replica, which is the optimal concurrency setting for the 2-core CPU limit scenario in Section 5.1. We start our experiments with all **light** requests for the first 450s, then we change the request type to **heavy** (see Figure 3(e) and 3(f)).

Sora achieves a relatively stable response time and goodput in a 12-minute experiment than that in the Kubernetes HPA case (see Figure 12(a)(i) and 12(b)(i)). For example, large response time fluctuations and goodput drops in the Kubernetes HPA case during the temporary overload phase after the request type changes (45s∼82s, 222s∼295s, and 508s∼578s). Before the request type changes from light to heavy at 451s, Figures 12(a)(i) and 12(b)(i) show that Sora achieves lower response time and higher goodput than Kubernetes HPA (e.g., 45s∼82s and 222s∼295s). This is because Sora can dynamically adjust the request connection pool size according to the number of Post Storage replicas. In contrast, Kubernetes HPA only adjusts the number of Post Storage replicas. The workload imbalance between existing replicas and newly-added replicas would cause a sub-optimal soft resource allocation among replicas and further degrade performance. Furthermore, we notice a large response time spike appears during period 508s∼578s in Figure 12(a)(i), and the current 50 connections allocation becomes the bottleneck (see Figure 12(a)(iii)). This is because serving heavy requests stresses downstream database services, making the Post Storage replicas route more requests to downstream services. The optimal connections to Post Storage should change from 10 to 30 (see Section 2.3). On the other hand, Figure 12(b)(iii) shows that Sora dynamically reallocates 120 connections for 4 Post Storage replicas, which can effectively stabilize the response time fluctuations caused by the system state drifting.

6 DISCUSSIONS

Scalability of Sora. Sora would work based on the assumption that our monitoring infrastructure can identify the critical services along the critical path in a large-scale system. We learn from FIRM [47] and implement an online distributed tracing but an offline data analysis on a centralized graph database, which stores the request/response timestamps of each microservice. Specifically, our SCG model estimates runtime service goodput and concurrency based on the request/response timestamps stored in the dedicated graph database, which does not add additional computational overhead to the target runtime system. We observed the collection of telemetry data and execution history graphs for critical service identification leads to a maximum CPU overhead of 5% and a 50ms computational overhead for all loads in our private testbed. However, Sora has two limitations that are subject to our future research: First, the efficiency of the centralized graph database determines the latency overhead of critical service extraction, which would limit the scalability of Sora. Second, the accuracy of critical service identification in the large-scale system would limit the effectiveness of Sora. The state-of-the-art ML approach in FIRM [47] admits a 93% accuracy in a large-scale system.

Applicability of Sora. Unlike hardware resources, soft resources usually have a large configuration space due to the heterogeneous service implementation (Section 2.1). We notice not all soft resources are suitable for being runtime reconfigured by autoscaling solutions. For example, exposing some internal application-specific soft resources (e.g., locks) requires a significant engineering effort by service providers, which may limit the applicability of autoscaler. Hence, we target server threads and connections since they are the most generic soft resources for heterogeneous microservices and can directly control the request processing concurrency of each microservice. Fortunately, most service providers explicitly expose the tuning knobs of these two types of soft resources so that administrators can easily reconfigure them by changing hosting server parameters (e.g., Tomcat threads pool) or third-party library parameters (e.g., JDBC connection pool).

7 RELATED WORK

Previous research on stabilizing performance fluctuations to meet strict SLOs for microservices-based applications in clouds can be broadly divided into three categories:

Autoscaling microservices-based applications adopt techniques to elasticize computing resources in clouds [27, 45, 48]. These techniques can be further categorized into four groups: (a) threshold-based (or rule-based) [31, 42, 44], (b) statistical profiling-based [17, 54], (c) analytical model-based [8, 13, 40, 76], and (d) machine learning-based [23, 39, 51, 66, 71]. For example, Kubernetes Horizontal Pod Autoscaler [7] adopts a straightforward rule-based approach to elasticize computing resources (e.g., CPU and memory) based on observed resource utilization (e.g., CPU utilization > 80%). SHOWAR [8] adopts basic ideas from control theory and kernel-level performance metrics to determine the optimal hardware resource allocations (e.g., CPU and memory). FIRM [47] and Fifer [20] combine statistical profiling and machine learning-based approaches to reprovision hardware resources to reduce SLO violations adaptively. However, none of these approaches can correspondingly scale the soft resources (e.g., threads and TCP connections) in microservices-based applications, which controls the concurrent use of hardware resources and could become significant sources of performance instability (see Section 2.3).

Analytical performance model for microservices-based applications has been widely used to estimate the microservice capacity and performance gains [22, 25, 29, 35, 41, 49, 52, 56, 59, 73]. For example, ATOM [18] leverages queueing network model to estimate the computational optimization to maximize the system performance with minimal CPU shares. Alibaba group [36] conducts in-depth anatomy of microservices call-dependency based on a production trace analysis to optimize microservice designs for SLOs. MicroRCA [67] can locate root causes of performance issues in microservices by correlating application performance symptoms with corresponding system resource utilization. However, these approaches require significant human effort and expert knowledge to conduct performance tuning. Their work inspires us to improve our work to be more agile when handling runtime system condition variations.

Experimental software reconfiguration approaches for microservices-based applications have been studied extensively [34, 38, 43, 68–70, 72]. For example, BestConfig [75] automates the configuration tuning for general systems by combining the divide-and-diverge sampling method and the recursive bound-and-search algorithm. ConScale [33] utilizes a Scatter-Concurrency-Throughput (SCT) model based on statistical correlations between each server's throughput and concurrency to quickly adapt the optimal soft resource configurations of key servers during the system scaling process. `Iter8` [61] adopts online Bayesian learning and multi-armed bandit algorithms to enable microservices-based applications tailored for SLOs in the cloud. Our work complements their work by integrating an online Scatter-Concurrency-Goodput (SCG) Model with more fine-grained runtime contextual information, which can capture each subtle change in system conditions and better adapt soft resource allocations for microservices-based applications.

8 CONCLUSION

We propose Sora, an online concurrency adapting framework that integrates latency constraints and fast concurrency adaptation for critical microservices and works together with other popular hardware-only autoscalers. Sora uses SCG, an online goodput-based model which collects fine-grained metrics extracted from various microservice instances to quickly determine an optimal soft resource setting for critical microservices during runtime. Our experiments using six real-world bursty workloads show that Sora can effectively reduce the tail response time of our microservices benchmark application at the 99th percentile by an average of 2.5× compared to the FIRM, and 1.5× to the state-of-the-art concurrency-aware system scaling strategy ConScale. Overall, Sora enables fast mitigation of user-perceived response time fluctuations by combining efficient hardware and soft resource provisioning, contributing to both high resource efficiency and high performance of modern cloud requirements.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd Dr. Arpan Gujarati for their feedback on improving this paper. This research has been partially funded by National Science Foundation by CNS (2000681), CNS (2245827), CNS (2212256), and contracts from Fujitsu Limited. Any opinions, findings, and conclusions are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies mentioned above.

REFERENCES

[1] Decomposing twitter: Adventures in service-oriented architecture. https://www.infoq.com/presentations/twitter-soa/.
[2] Mean absolute percentage error. https://en.wikipedia.org/wiki/Mean_absolute_percentage_error.
[3] mongodb. https://www.mongodb.com.
[4] Neo4j:native graph database. https://github.com/neo4j/neo4j.
[5] Tony mauro. adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.
[6] Sock shop microservice demo application. https://microservices-demo.github.io/, 2016.
[7] Kubernetes horizontal pod auto-scaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2019.
[8] Baarzi, A. F., and Kesidis, G. SHOWAR: Right-sizing and efficient scheduling of microservices. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA,*

USA, November 1 - 4, 2021, C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 427–441.

[9] Benesty, J., Chen, J., Huang, Y., and Cohen, I. Pearson correlation coefficient. In *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[10] Chiba, T., Nakazawa, R., Horii, H., Suneja, S., and Seelam, S. Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes. In *2019 IEEE International Conference on Cloud Engineering (IC2E)* (2019), IEEE, pp. 168–178.

[11] Consortium, O. Rubbos: Bulletin board benchmark. http://jmob.ow2.org/rubbos.html, 2005.

[12] container, L. Infrastructure for container projects. https://linuxcontainers.org/.

[13] Cusack, G., Nazari, M., Goodarzy, S., Hunhoff, E., Oberai, P., Keller, E., Rozner, E., and Han, R. Escra: Event-driven, Sub-second Container Resource Allocation. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)* (July 2022), pp. 313–324.

[14] Docker. Docker. https://www.docker.com/.

[15] Einav, Y. Amazon found every 100ms of latency cost them 1 https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/.

[16] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., 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), pp. 3–18.

[17] Gandhi, A., Harchol-Balter, M., Raghunathan, R., and Kozuch, M. A. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS) 30*, 4 (2012), 14.

[18] Gias, A. U., Casale, G., and Woodside, M. Atom: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), IEEE, pp. 1994–2004.

[19] golang. Go official website. https://golang.org/.

[20] Gunasekaran, J. R., Thinakaran, P., Nachiappan, N. C., Kandemir, M. T., and Das, C. R. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference* (New York, NY, USA, 2020), Middleware '20, Association for Computing Machinery, p. 280–295.

[21] Guo, X., Peng, X., Wang, H., Li, W., Jiang, H., Ding, D., Xie, T., and Su, L. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 1387–1397.

[22] Huang, L., and Zhu, T. Tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 76–91.

[23] Hwang, C., Kim, T., Kim, S., Shin, J., and Park, K. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (2021), pp. 721–739.

[24] Jaeger. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.

[25] Jindal, A., Podolskiy, V., and Gerndt, M. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (2019), pp. 25–32.

[26] Jolokia. Jolokia official website. https://jolokia.org/.

[27] Jyothi, S. A., Curino, C., Menache, I., Narayanamurthy, S. M., Tumanov, A., Yaniv, J., Mavlyutov, R., Goiri, I., Krishnan, S., Kulkarni, J., et al. Morpheus: Towards automated {SLOs} for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 117–134.

[28] Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles* (2017), pp. 34–50.

[29] Kannan, R. S., Subramanian, L., Raju, A., Ahn, J., Mars, J., and Tang, L. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, Mar. 2019), EuroSys '19, Association for Computing Machinery, pp. 1–16.

[30] Kubernetes. Kubernetes. https://kubernetes.io/.

[31] Kwan, A., Wong, J., Jacobsen, H.-A., and Muthusamy, V. Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), pp. 80–90.

[32] Liu, H., Zhang, J., Shan, H., Li, M., Chen, Y., He, X., and Li, X. Jcallgraph: tracing microservices in very large scale container cloud platforms. In *International Conference on Cloud Computing* (2019), Springer, pp. 287–302.

[33] Liu, J., Zhang, S., Wang, Q., and Wei, J. Mitigating large response time fluctuations through fast concurrency adapting in clouds. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2020), IEEE, pp. 368–377.

[34] Liu, J., Zhang, S., Wang, Q., and Wei, J. Coordinating fast concurrency adapting with autoscaling for slo-oriented web applications. *IEEE Transactions on Parallel and Distributed Systems 33*, 12 (2022), 3349–3362.

[35] Liu, L., Wang, H., Wang, A., Xiao, M., Cheng, Y., and Chen, S. Mind the gap: Broken promises of CPU reservations in containerized multi-tenant clouds. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 243–257.

[36] Luo, S., Xu, H., Lu, C., Ye, K., Xu, G., Zhang, L., Ding, Y., He, J., and Xu, C. Characterizing microservice dependency and performance: Alibaba trace analysis. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 412–426.

[37] Mahgoub, A., Wood, P., Ganesh, S., Mitra, S., Gerlach, W., Harrison, T., Meyer, F., Grama, A., Bagchi, S., and Chaterji, S. Rafiki: a middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (2017), pp. 28–40.

[38] Maji, A. K., Mitra, S., Zhou, B., Bagchi, S., and Verma, A. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference* (2014), pp. 277–288.

[39] Mao, H., Alizadeh, M., Menache, I., and Kandula, S. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks* (2016), pp. 50–56.

[40] Mirhosseini, A., Elnikety, S., and Wenisch, T. F. Parslo: A gradient descent-based approach for near-optimal partial SLO allotment in microservices. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 442–457.

[41] Mirhosseini, A., West, B. L., Blake, G. W., and Wenisch, T. F. Express-lane scheduling and multithreading to minimize the tail latency of microservices. In *2019 IEEE International Conference on Autonomic Computing (ICAC)* (2019), IEEE, pp. 194–199.

[42] Mittal, V., Qi, S., Bhattacharya, R., Lyu, X., Li, J., Kulkarni, S. G., Li, D., Hwang, J., Ramakrishnan, K. K., and Wood, T. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 168–181.

[43] Mvondo, D., Barbalace, A., Tchana, A., and Muller, G. Tell me when you are sleepy and what may wake you up! In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 562–569.

[44] Netto, M. A., Cardonha, C., Cunha, R. L., and Assuncao, M. D. Evaluating auto-scaling strategies for cloud computing environments. In *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems* (2014), IEEE, pp. 187–196.

[45] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 361–378.

[46] Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., and Josuttis, N. Microservices in practice, part 1: Reality check and service design. *IEEE Annals of the History of Computing 34*, 01 (2017), 91–98.

[47] Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K. {FIRM}: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 805–825.

[48] Qu, C., Calheiros, R. N., and Buyya, R. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR) 51*, 4 (2018), 73.

[49] Rahman, J., and Lama, P. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)* (2019), IEEE, pp. 200–210.

[50] Ren, R., Ma, J., Sui, X., and Bao, Y. D2p: a distributed deadline propagation approach to tolerate long-tail latency in datacenters. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (2014), pp. 1–6.

[51] Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., and Wilkes, J. Autopilot: Workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, Apr. 2020), EuroSys '20, Association for Computing Machinery, pp. 1–16.

[52] Samanta, A., Jiao, L., Mühlhäuser, M., and Wang, L. Incentivizing Microservices for Online Resource Sharing in Edge Clouds. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 420–430.

[53] Satopaa, V., Albrecht, J., Irwin, D., and Raghavan, B. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops* (2011), IEEE, pp. 166–171.

[54] Sharma, U., Shenoy, P., Sahu, S., and Shaikh, A. A cost-aware elasticity provisioning system for the cloud. In *2011 31st International Conference on Distributed Computing Systems* (2011), IEEE, pp. 559–570.

[55] Somashekar, G., and Gandhi, A. Towards optimal configuration of microservices. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (2021), pp. 7–14.

[56] Song, W., Xiao, Z., Chen, Q., and Luo, H. Adaptive Resource Provisioning for the Cloud Using Online Bin Packing. 2647–2660.

[57] Spring. Spring boot overview. https://spring.io/projects/spring-boot.

[58] Sriraman, A., and Wenisch, T. F. μtune: Auto-tuned threading for {OLDI} microservices. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 177–194.

[59] Tennage, P., Perera, S., Jayasinghe, M., and Jayasena, S. An analysis of holistic tail latency behaviors of java microservices. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2019), IEEE, pp. 697–705.

[60] Thomas, W., and Colin, K. gnuplot homepage. http://www.gnuplot.info/, 2019.

[61] Toslali, M., Parthasarathy, S., Oliveira, F., Huang, H., and Coskun, A. K. Iter8: Online experimentation in the cloud. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 289–304.

[62] Virtuozzo. Open source container-based virtualization for linux. https://openvz.org/.

[63] VMware. Vmware esxi: The purpose-built bare metal hypervisor. https://www.vmware.com/products/esxi-and-esx.html, 2019.

[64] Wang, Q., Chen, H., Zhang, S., Hu, L., and Palanisamy, B. Integrating concurrency control in n-tier application scaling management in the cloud. *IEEE Transactions on Parallel and Distributed Systems 30*, 4 (2018), 855–869.

[65] Wang, Q., Zhang, S., Kanemasa, Y., Pu, C., Palanisamy, B., Harada, L., and Kawaba, M. Optimizing n-tier application scalability in the cloud: A study of soft resource allocation. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 4*, 2 (2019), 1–27.

[66] Wang, Z., Zhu, S., Li, J., Jiang, W., Ramakrishnan, K. K., Zheng, Y., Yan, M., Zhang, X., and Liu, A. X. DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing* (New York, NY, USA, Nov. 2022), SoCC '22, Association for Computing Machinery, pp. 16–30.

[67] Wu, L., Tordsson, J., Elmroth, E., and Kao, O. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium* (2020), IEEE, pp. 1–9.

[68] Xiao, Z., Song, W., and Chen, Q. Dynamic Resource Allocation Using Virtual Machines for Cloud Computing Environment. 1107–1117.

[69] Xu, T., Jin, X., Huang, P., Zhou, Y., Lu, S., Jin, L., and Pasupathy, S. Early detection of configuration errors to reduce failure damage. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 619–634.

[70] Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., and Pasupathy, S. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), pp. 244–259.

[71] Yang, Z., Nguyen, P., Jin, H., and Nahrstedt, K. Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), IEEE, pp. 122–132.

[72] Zhang, B., Van Aken, D., Wang, J., Dai, T., Jiang, S., Lao, J., Sheng, S., Pavlo, A., and Gordon, G. J. A demonstration of the ottertune automatic database management system tuning service. *Proceedings of the VLDB Endowment 11*, 12 (2018), 1910–1913.

[73] Zhang, S., Wang, Q., Kanemasa, Y., Liu, J., and Pu, C. Doublefacead: A new datastore driver architecture to optimize fanout query performance. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 430–444.

[74] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q., and He, C. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 683–694.

[75] Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K., and Yang, Y. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), pp. 338–350.

[76] Zhu, Z., Bi, J., Yuan, H., and Chen, Y. Sla based dynamic virtualized resources provisioning for shared cloud data centers. In *2011 IEEE 4th International Conference on Cloud Computing* (2011), IEEE, pp. 630–637.

[77] Zipkin. Zipkin: A distributed system. https://zipkin.io/.