μConAdapter: Reinforcement Learning-based Fast Concurrency Adaptation for Microservices in Cloud

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

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

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

ABSTRACT

Modern web-facing applications such as e-commerce comprise tens or hundreds of distributed and loosely coupled microservices that promise to facilitate high scalability. While hardware resource scaling approaches [28] have been proposed to address response time fluctuations in critical microservices, little attention has been given to the scaling of soft resources (e.g., threads or database connections), which control hardware resource concurrency. This paper demonstrates that optimal soft resource allocation for critical microservices significantly impacts overall system performance, particularly response time. This suggests the need for fast and intelligent runtime reallocation of soft resources as part of microservices scaling management. We introduce μConAdapter, an intelligent and efficient framework for managing concurrency adaptation. It quickly identifies optimal soft resource allocations for critical microservices and adjusts them to mitigate violations of service-level objectives (SLOs). μConAdapter utilizes fine-grained online monitoring metrics from both the system and application levels and a Deep Q-Network (DQN) to quickly and adaptively provide optimal concurrency settings for critical microservices. Using six realistic bursty workload traces and two representative microservices-based benchmarks (*SockShop* and *SocialNetwork*), our experimental results show that μConAdapter can effectively mitigate large response time fluctuation and reduce the tail latency at the 99th percentile by 3× on average when compared to the hardware-only scaling strategies like Kubernetes Autoscaling and FIRM [28], and by 1.6× to the state-of-the-art concurrency-aware system scaling strategy like ConScale [21].

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624980

CCS CONCEPTS

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

KEYWORDS

Microservices, Auto-scaling, Soft Resource, Scalability

ACM Reference Format:
Jianshu Liu, Shungeng Zhang, and Qingyang Wang. 2023. μConAdapter: Reinforcement Learning-based Fast Concurrency Adaptation for Microservices in Cloud. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624980

1 INTRODUCTION

Modern web-facing applications have widely adopted microservices due to their superior scalability. Industry giants such as Amazon [2], Twitter [1], and Netflix [3] have migrated their core systems and architectures from traditional monolithic designs to microservices. Microservices are designed to efficiently handle naturally bursty workloads while meeting stringent Service-Level Objectives (SLOs), such as maintaining bounded response times. For instance, `Amazon.com` reported that every additional 100 milliseconds in page loading can lead to a 1% loss in sales [7]. An important feature of the microservices-based architecture is that the scalable fine-grained component microservices [26] can provide greater performance control by adding system resources (e.g., vCPU) only to the sections that need extra capacity.

Existing approaches have made notable efforts in hardware resource management to handle the variance in the critical path of microservices, such as FIRM [28], which focuses on the shared low-level resource contention. However, little attention has been given to scaling soft resources (e.g., threads or database connections) that control hardware resource concurrency. Previous studies [21, 34] revealed that the mismatch between the soft and hardware resources is an important factor contributing to SLO violations. For example, Figure 1 shows a microservice application with FIRM [28] encounters unexpected response time spikes over the scaling phases. This is caused by the over-allocation of the thread

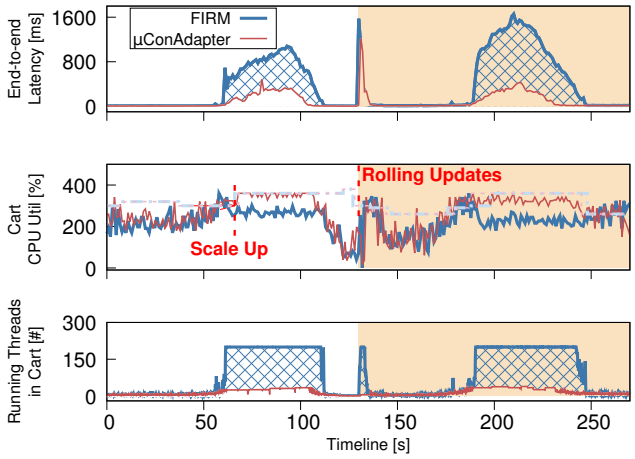


Figure 1: Large latency spikes from thread pool over-allocation. μConAdapter complements FIRM [28] with intelligent concurrency adaptations during resource scaling and microservice updates.

pool inside the critical microservices. Recent studies [20, 21] introduced the concurrency-aware system scaling solutions (e.g., ConScale) that employ online statistical models to efficiently adjust soft resource allocations of critical services during system scaling. However, the dynamic nature of microservice environments, due to frequent runtime changes (e.g., rolling updates of microservices in Figure 1), may cause recurring human efforts for offline model reconstruction. The online models face limitations in estimating short-term optimal allocations based on past observations, which may lead to a degraded performance caused by an unstable adaptation process, as discussed in Section 3.1.

In this paper, we propose μConAdapter, an online Reinforcement Learning (RL)-based concurrency adaptation management framework designed to quickly recommend and re-allocate soft resources (e.g., server threads and database connections) for critical microservices to mitigate SLO violations. The RL approach models the concurrency adaptation as a Markov decision process and recommends long-term optimal actions with a tight feedback loop. In particular, we use Deep Q-Network (DQN), a value-based method effective in solving problems with discrete action spaces, to learn directly from the actual workload. μConAdapter leverages fine-grained online monitoring metrics (e.g., resource utilization, request rate, throughput, response time, and concurrency) to describe the runtime system state and workload characteristics and feeds them to DQN for optimal soft resources estimation. This is primarily because soft resources, such as server threads and database connections, control the sharing of hardware resources in microservices through concurrency. A conservative allocation (e.g., too small a thread pool) often creates software bottlenecks that limit overall system throughput. In contrast, a liberal allocation often wastes hardware resources such as CPU and memory. Figure 1 demonstrates the need for our μConAdapter. It enables fast and intelligent runtime adaptation of the thread pool size for a critical microservice when scaling the *SockShop* [4] microservices benchmark application to achieve better performance.

The first contribution of the paper is an empirical study based on the microservices benchmark, *SockShop* [4]). This study confirms that optimal concurrency settings can significantly vary under different system state changes in a microservices-based web application. For example, we show that the optimal server thread pool size for one of the microservices (e.g., *Cart* service) exhibits a superlinear increase from 5 to 30 after the CPU limit scales up from a 2-core to a 4-core configuration (see Figure 3(a)). We also observed that a sub-optimal thread allocation, given the same CPU limit, could degrade the maximum achievable throughput of a *Cart* service instance by up to 50% (see Figure 3(a)).

Our second contribution is a Reinforcement Learning (RL)-based approach that enables fast and smart runtime concurrency adaptation for microservices in clouds. We feed the runtime system and workload information (e.g., fine-grained system and application-level metrics) to a Deep Q-Network model, which makes frequent concurrency adaptation decisions (Section 3). We further propose a mechanism to speed up the training process of our DQN model through historical fine-grained sampling metrics. By continuously learning from the rewards of various concurrency settings, the RL-based model can avoid the low-efficiency problem for model convergence and quickly adapt to optimal concurrency.

The third contribution is the design and implementation of the μConAdapter framework, which leverages our RL-based agent to coordinate the provisioning of both hardware and soft resources in microservices (Section 4). Through extensive experiments using six realistic bursty workload traces [9] on two representative microservices-based benchmark applications (i.e., SockShop and SocialNetwork), we demonstrate that μConAdapter can effectively alleviate large response time fluctuations and reduce the 99th percentile latencies by 3× on average. This is compared to hardware-only scaling strategies like Kubernetes Autoscaling [16] and FIRM [28], and it offers 1.6× improvement over the state-of-the-art concurrency-aware scaling, ConScale [21].

The rest of the paper is organized as follows. Section 2 presents experimental results showing that sub-optimal soft resource allocation leads to significant performance degradation. Section 3 introduces our RL-based model. Section 4 illustrates the design and implementation of our μConAdapter framework. Section 5 shows the evaluation under six realistic workloads. Section 6 and Section 7 summarize the limitations and related work, and Section 8 concludes the paper.

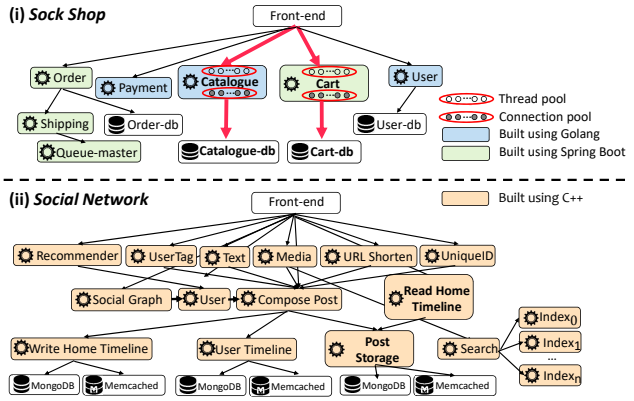


Figure 2: Service dependency graphs of *SockShop* and *SocialNetwork* benchmark applications.

2 BACKGROUND AND MOTIVATION

2.1 Soft Resource Adaptation in Microservice

Soft Resources. Unlike well-defined hardware resources (such as CPU, memory, disk, and network) in performance evaluation studies, *soft resources* refer to system software components that consume hardware resources. For example, threads consume CPU and memory while TCP connections multiplex network I/O. For an Internet server, threads and network connections (e.g., database connections or AJP connections, typically on top of TCP) are the two most important soft resources because they control the request processing concurrency level and increase the hardware utilization efficiency through the sharing of hardware resources.

Runtime Soft Resource Reallocation. Many service providers explicitly expose the tuning knobs of middleware-level soft resources (e.g., server threads and connections) that can be easily reconfigured by changing hosting server parameters (e.g., Tomcat thread pool) or third-party library parameters (e.g., JDBC connection pool). For example, we can adjust the thread pool size for the SpringBoot-based service through remote JMX access via *Jolokia* and manage the database connection pool size in the Golang-based service through a manually extended service by calling APIs in the Golang package "database/sql". The cost of changing configurations can be low, with many service implementations offering runtime-resizable APIs for a graceful exit. However, there may exist other internal application-specific soft resources (e.g., locks). Tuning such internal soft resources requires additional engineering effort from cluster orchestration, which is beyond this paper's scope.

2.2 Experimental Setup

We adopt two open-sourced microservices benchmarks in Figure 2: (i) SockShop [4], and (ii) SocialNetwork from the DeathStarBench [8] benchmark. SockShop is an e-commerce website with 11 unique microservices, which allows customers to navigate and purchase different socks. SocialNetwork (with 36 unique microservices) implements a broadcast-style social network with uni-directional follow relationships whereby users can publish and read social media posts. We use the RUBBoS workload generator [6] to simulate a number of concurrent users accessing the target microservices, where the request rate follows a Poisson distribution.

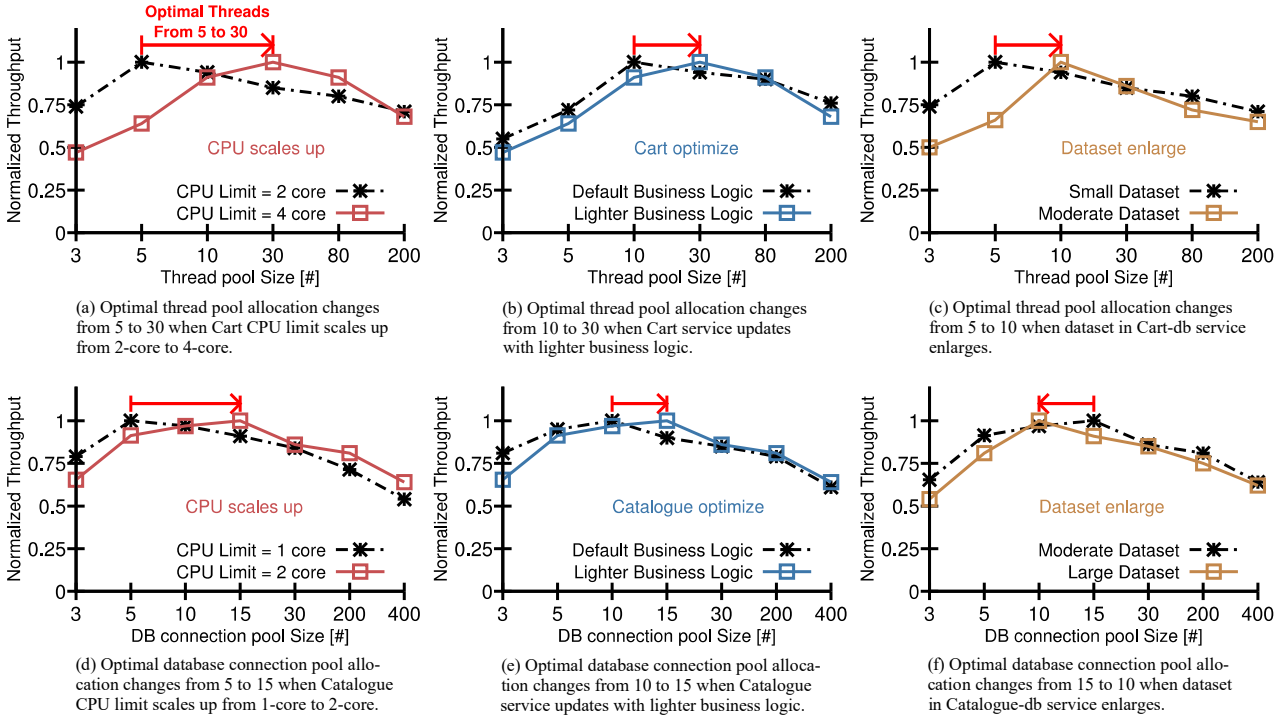
We conducted experiments in our private VMware ESXi cluster, which consists of 6 bare-metal servers equipped with two Intel Xeon E5-2603v3 processors and 16GB of RAM. We deployed 18 VMs in the cluster, and each VM was configured with 4 vCPUs, 4GB RAM, and 30GB disk space. Furthermore, we set up a Kubernetes cluster for container orchestration and deployed containers uniformly among VMs. In this paper, we use the terms "pod" and "container" interchangeably since we use a standard one-container-per-pod model. Each container runs one component microservice. In the following motivation experiments, we evaluate two representative soft resources from SockShop: the thread pool in a SpringBoot-based service *Cart* and the database connection pool in a Golang-based service *Catalogue*.

2.3 Performance Degradation with Sub-Optimal Concurrency Settings

In this section, we show our empirical study on three runtime system condition changes that affect the optimal concurrency setting in a microservices-based web application.

1) Hardware Resource Scaling. The vertical scaling (e.g., adding or removing # of CPU cores) would affect the optimal concurrency setting of a microservice instance. Figure 3(a) shows that the optimal thread pool allocation shifts from 5 to 30 when the CPU limit for *Cart* service scales up from 2-core to 4-core. This is because the original optimal concurrency setting (i.e., five server threads) becomes an under-allocation and cannot fully utilize four CPU cores after the system scaling. We have observed consistent experimental results about the shifts of optimal connection pool allocation in *Catalogue* service as the CPU limit scales up, as shown in Figure 3(d). These experimental results indicate that hardware resource scaling would cause the original optimal concurrency setting to be sub-optimal, resulting in performance degradation.

2) Upgrading of Microservices Business Logic. Furthermore, we explore the impact of the upgrades to the business logic for serving the same type of requests of a microservice instance. For example, we update *Cart* and *Catalogue* to employ an optimized business logic with fewer computations (i.e., less service time). Figure 3(b) shows that the optimal thread pool allocation *increases* from 10 to 30 after applying the optimized business logic for *Cart* service. This



(a) Optimal thread pool allocation changes from 5 to 30 when Cart CPU limit scales up from 2-core to 4-core.

(b) Optimal thread pool allocation changes from 10 to 30 when Cart service updates with lighter business logic.

(c) Optimal thread pool allocation changes from 5 to 10 when dataset in Cart-db service enlarges.

(d) Optimal database connection pool allocation changes from 5 to 15 when Catalogue CPU limit scales up from 1-core to 2-core.

(e) Optimal database connection pool allocation changes from 10 to 15 when Catalogue service updates with lighter business logic.

(f) Optimal database connection pool allocation changes from 15 to 10 when dataset in Catalogue-db service enlarges.

Figure 3: Performance variation at increasing soft resource allocation (e.g., thread pool and database connection pool) for *Cart* and *Catalogue* service in *Sock Shop* benchmark application. Figures (a) and (d) show that scaling up # of CPU cores could change the optimal soft resource allocation on microservices, Figures (b) and (e) show the case of changing the business logic of microservices, and Figures (c) and (f) show system state variation case.

is because the degrees of computation for the business logic for serving the same type of requests are proportional to the CPU resource consumption – the fewer computations for the business logic, the less CPU resource consumed. Thus, the original optimal concurrency setting (i.e., ten threads) becomes an under-allocation and cannot fully utilize all CPU resources after applying the optimized business logic. Consistent results are observed in Figure 3(e) for *Catalogue*.

3) Drifting of System State. The system state drift also affects the optimal concurrency setting of a component microservice by affecting the degrees of computation for the business logic for serving the same type of requests. For example, the system state of the backend microservices could drift over time due to continuous dataset updates, which leads to variations of the service time of requests in upstream microservices accordingly. In this set of experiments, we initially employed the original dataset for *Cart-db* and *Catalogue-db* services and then manually enlarged the original dataset. Figure 3(c) shows that the optimal thread pool in *Cart* service *increases* from 5 to 10 after we enlarge the dataset in *Cart-db* service. An interesting observation is that Figure 3(f) shows the opposite results, where the optimal database connection pool in *Catalogue decreases* from 15 to 10 after enlarging the dataset in *Catalogue-db*. Such experimental results show that the effect of system state drift on different microservices can be heterogeneous due to complex dependencies among these microservices.

Our three empirical observations demonstrate that the runtime system condition changes have a significant impact on the optimal concurrency in different microservices. These runtime system condition changes, especially the hardware resource scaling, are common for microservices-based web applications in cloud environments due to the naturally bursty workload. Hence, service providers need an online approach to quickly and accurately identify the optimal concurrency settings according to various system conditions.

3 RL-BASED MODEL FOR OPTIMAL CONCURRENCY ADAPTATION

This section proposes a reinforcement learning-based model for quickly adapting optimal concurrency settings for critical microservice instances, which can address the limitations of existing statistical approaches for online concurrency adaptation (Section 3.1). Our model collects fine-grained contextual metrics (e.g., system conditions) and inputs them into a Deep-Q-Network (DQN), which recommends the optimal concurrency setting to achieve the highest reward (Section 3.2). We

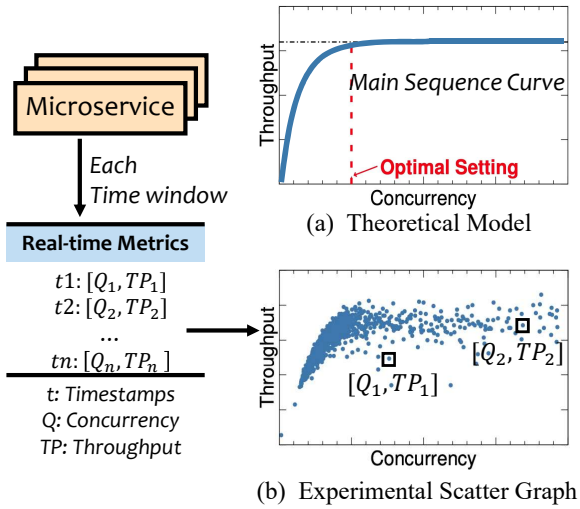


Figure 4: An online statistical model estimates optimal concurrency setting based on relationships between real-time service throughput and concurrency.

further introduce a mechanism to accelerate the training process of our DQN-based model through the use of fine-grained sampling metrics (Section 3.3).

3.1 Statistical Model Limitations

Existing approaches for determining optimal concurrency mainly adopt statistical methods involving static offline models [12, 34] and dynamic online models [21, 39].

Static offline models identify optimal concurrency settings using a brute-force search to model the relationships between performance metrics based on queuing theory. For example, Figure 4(a) characterizes the theoretical relationship (main sequence curve) between a microservice instance's throughput and concurrency based on the classic Utilization Law [17]. The process of training model parameters is very time-consuming since we need to tune concurrency settings step-by-step and run corresponding experiments for each step (e.g., from 3 to 200 in Figure 3(a)). In addition, model reconstruction and retraining are required for the static offline models when the system conditions change at runtime (e.g., hardware resource reallocation and change in system state such as dataset [34]). Thus, offline models cannot quickly adapt to the optimal concurrency setting for latency-sensitive microservices applications, which usually have strict SLOs.

The dynamic online model [21] solves such time-consuming problems by building and revising the performance model during runtime. Unlike the brute-force search in offline models, the online model estimates the optimal resource allocations for each service based on the correlation among the real-time fine-grained monitoring metrics (e.g., throughput and concurrency) within a short time window (e.g., 3 minutes). For example, Figure 4(b) uses a scatter graph to characterize the correlation between a service's throughput and concurrency, measured at a 100ms time granularity during a 3-minute time window.

However, such a correlation model suffers from two limitations. First, extracting the main sequence curve from a scatter graph is a non-trivial task since many factors could affect the quality of the scatter graph. For example, an inappropriate time window could lead to large variations in the concurrency-throughput pairs due to the complex system dynamics, making the main sequence curve extraction extremely difficult from the scatter graph. Second, the online model recommends the optimal concurrency setting every once within a pre-defined time window (e.g., 3 minutes), which is still considered to have a coarse time granularity for the bursty workload in microservices [15]. For example, the online model-based ConScale [21] framework only tunes the optimal soft resource allocation every 3 minutes as model correlation requires sufficient real-time metrics. Such coarse concurrency adaptation cannot guarantee SLO requirements for microservices because system condition changes (e.g., hardware resource reallocation and system state updates) are far more frequent in microservices-based applications [37].

3.2 Concurrency Adaptation Using RL

Reinforcement Learning (RL)-based model inherits the benefits of online models and further resolves their limitations. First, the RL model improves the concurrency estimation by adopting the historical data replay mechanism (e.g., Experience Replay [24]) and involving more contextual information (e.g., system conditions and resource utilization) as model input. The historical data replay mechanism can mitigate large variations in updates by training the parameters with a minibatch consisting of randomly selected historical data, enabling a more stable learning procedure to provide valid estimations continuously. Moreover, the RL model with more contextual system information provides a more accurate estimation to handle system condition variations than does a throughput-driven correlation model. This is because the RL model can characterize the runtime environment more precisely by comprehensively considering throughput, resource utilization, and changes in system conditions.

Second, the RL model can provide frequent concurrency adaptations (e.g., 1s) based on a tight feedback loop. The RL model determines the optimal actions directly based on the states of the environment without relying on the correlations between real-time metrics over a time window (e.g., 3 min). Therefore, the RL model saves time during the data collection phase to provide fast concurrency adaptation to handle

dynamic system conditions in a microservices-based system. Besides, the RL model allows direct learning from the actual workload without human intervention, which reduces the significant efforts of experts compared to statistical models.

What is RL? RL refers to a *trial-and-error* learning mechanism. Concretely, the RL model learns to solve a sequential decision-making problem by interacting with the environment. The model observes a *state* ($s_t \in S$) from the environment at each time step t. Then it performs an *action* ($a_t \in A$) that can maximize the action-value function $Q(s_t, a_t; \theta)$, which records the estimated reward of available actions mapped to the current state. After the action is taken at t+1, the model observes a *reward* ($r_t \in R$) given by a reward function $r(s_t, a_t)$ and forms a tuple (*transition*) (s_t, a_t, r_t, s_{t+1}). The model's goal is to optimize the action-value function to maximize the expected *cumulative discounted reward*, where the return from a state is defined as $\Sigma_{k=0}^{T} \gamma^k r_{t+k}$. The discount factor $\gamma \in (0, 1]$ penalizes the predicted future rewards.

Deep Q-Learning Algorithm. In particular, we utilize the deep Q-learning algorithm by adopting a DQN model for optimal concurrency setting adaptation. The DQN model directly learns from high-dimensional observed data using a deep neural network to approximate the Q-value function and performs actions with the highest expected Q-value. Compared to other dynamic online models (e.g., Bayesian Optimization (BO) [18], Contextual Bandits (CB) [31], and Deep Deterministic Policy Gradient (DDPG) [28]), DQN provides two distinct advantages.

- DQN models the concurrency adaptation as a Markov decision process and optimizes for a long-term cumulative reward in dynamic microservice environments. BO and CB target short-term optimal actions based on past observations without considering future states, which may cause an unstable learning process and incur SLO violations (see Figure 7(c)).
- DQN is a value-based method that can effectively solve problems with a discrete action space by directly evaluating the Q-value for each concurrency setting. DDPG, based on an actor-critic approach, adds a policy-based actor-network, which involves more hyperparameters in training and would slow down the learning process (see Figure 7(d)).

Algorithm 1 shows the pseudocode of the training algorithm. To make our RL model explore better actions during the training process, we add noise with a probability of ϵ to the action selected based on the Q-value (line 8). Such a strategy makes the RL model exploit the rational actions over the entire training process, as it can avoid bad trials for allocating too much concurrency due to random strategy. On the other hand, to eliminate oscillations of the update effect, DQN builds a target Q network $\hat{Q}(s_t, a_t; \hat{\theta})$ which is a copy of

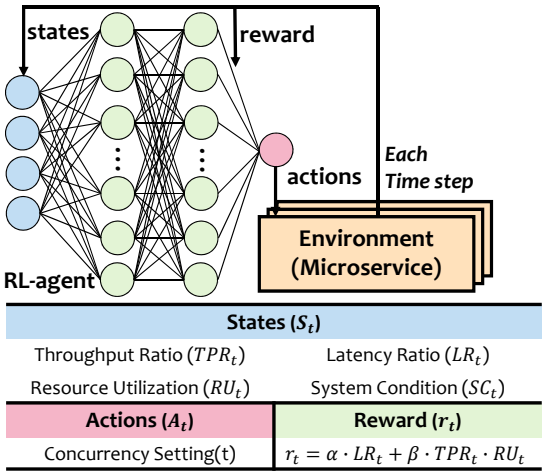


Figure 5: Architecture of RL model with a Deep Q-Network (DQN) for estimating optimal concurrency setting in a microservice instance.

the value function $Q(s_t, a_t; \theta)$. We synchronize the estimated network with the target network every C steps to stabilize learning. The value function $Q(s_t, a_t; \theta)$ with parameter θ and its corresponding loss function $L(\theta)$ are defined as:

$$Q(s_t, a_t; \theta) = \mathbb{E}[r(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}; \theta)] \quad (1)$$

$$L(\theta) = \frac{1}{N} \Sigma_i (r_i + \gamma \hat{Q}(s_{i+1}, a_{i+1}; \hat{\theta})) - Q(s_i, a_i; \theta))^2 \quad (2)$$

Problem Formulation. We formulate the concurrency settings (i.e., soft resource allocation) adaptation as a sequential decision-making problem that can be solved by the above RL framework. At each time step t, we measure resource utilization (e.g., CPU utilization) and request rate, throughput, response time, and current system conditions, including computational complexity of service business logic and dataset size for database services (Section 2.3). The RL model calculates the states listed in Figure 5 with these measurements. *Throughput Ratio* (TPR_t) is defined as the current throughput to request rate ratio. *Latency Ratio* (LR_t) is defined as SLO Latency/Response Time. If no messages arrive or the response time is zero, it is assumed that there is no SLO violation ($LR_t = 1$). *Resource Utilization* (RU_t) is defined as the ratio of critical hardware resource utilization to its limit (e.g., $CPU_t = CPUutil./CPUlimit$). *System Condition* ($SC_t$) is an array to summarize all microservices' underlying hardware resource limits and deployment within an execution path. For example, the system condition for a "*Cart*" service can be labeled as [code version:"*Cart*:Light-v1", dataset:"*Cart-db*:User-2K"]. We extract this information by referring to the container status and further encode it for model input without manual labeling. A better system conditions modeling would further improve the performance of our RL model in the future. Reward function design has non-trivial implications on the concurrency adaptation policy. The ultimate

Algorithm 1 Deep Q-learning with Experience Replay

Init replay memory D
Init action-value function Q with random weights θ
Init target action-value function $\hat{Q}$ with weights $\hat{\theta}$
for *episode* = 1, M **do**
  Receive initial observation state s_1
  for $t = 1, T$ **do**
    Select action $a_t = argmax_a Q(s_t, a; \theta)$
    With probability ϵ set Noise $N = N(a_t)$
    Execute action $a_t + N(a_t)$
    Observe new state s_{t+1}
    Generate and Store transition(s_t, a_t, r_t, s_{t+1}) in D
    Sample N transitions (s_i, a_i, r_i, s_{i+1}) from D
    Set $y_i = r_i + \gamma max_{a'} \hat{Q}(s_{i+1}, a'; \hat{\theta})$
    Perform a gradient descent step $(y_i - Q(s_i, a_i; \theta))^2$
    Every C steps reset $\hat{Q} = Q$
  end for
end for

goal of our RL model is to learn to select the optimal action that minimizes the latency (i.e., $min_{\pi_t} LR_t$) while maximizing the throughput under high resource utilization (i.e., $max_{\pi_t} TPR_t \cdot RU_t$). Based on both objectives, the reward function is defined as $r_t = \alpha \cdot LR_t + \beta \cdot TPR_t \cdot RU_t$.

3.3 Speed up via Fine-grained Sampling Metrics

DQN adopts the ϵ-greedy exploration strategy to choose a random action with a probability of ϵ. However, it may suffer from data inefficiency in environments with sparse rewards, which slows down the learning process [10]. Besides traditional boosting approaches (e.g., prioritized experience replay), we propose utilizing fine-grained sampling (e.g., 100ms) reward-related metrics to generate more valid transitions within limited physical time steps. The metrics include request rate, throughput, response time, and concurrency.

Our idea is similar to the approaches that use a multi-agent DQN to enhance the learning process by parallel training [33]. In contrast, we adopt a single agent and regard the measured concurrency as the action trials from other agents to generate multiple transitions and store them in the memory buffer within a physical time step. Such an approach inherits the knowledge derived from the existing online correlation model (Figure 4(b)) [21]. The correlations between concurrency and performance reward serve as the prior knowledge and help our RL agent quickly learn from the experience. Moreover, using fine-grained concurrency as actions can mitigate the reward fluctuations due to system state variations during online exploration. Our approach enables the DQN to perform 6 × faster than the vanilla DQN under a single system condition training scenario. We will show experimental validations in Section 5.2.

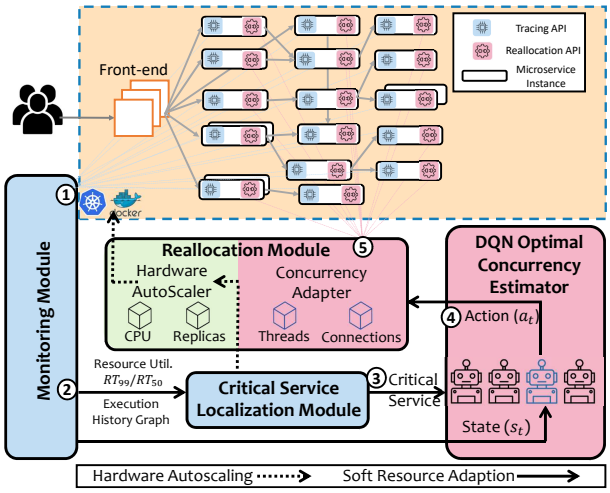


Figure 6: μConAdapter framework for coordinating optimal concurrency adaptation with hardware scaling.

4 μCONADAPTER FRAMEWORK

So far, we have described our RL model that applies runtime optimal concurrency adaptation to a microservice. Our experimental results show that hardware-only scaling cannot handle performance degradation caused by inappropriate soft resource allocation (Figure 1); thus, runtime soft resource adaptation management is required to complement the hardware-only autoscalers to achieve better performance. This section presents our μConAdapter framework, which integrates the RL-based concurrency adaptation (Section 3) to work with a hardware-only autoscaler. Figure 6 shows the four main components of μConAdapter: Monitoring Module, Critical Service Localization Module, DQN Optimal Concurrency Estimator, and Reallocation Module.

4.1 Module Design

Monitoring Module collects both application- and system-level metrics (e.g., throughput and CPU). We use distributed tracing and public monitoring tools to implement the monitoring module. Distributed tracing is a popular method to monitor microservices by recording the arrival and departure timestamps of a request as a span within a microservice. We implement an OpenTracing-compliant tracing module inside each microservice to collect request spans and generate application-level metrics by processing these spans. Additionally, the monitoring module also extracts system-level metrics (e.g., container resource utilization) via cAdvisor.

Critical Microservice Localization Module is responsible for identifying the bottlenecked microservice along an execution path in the system triggered by an HTTP request. This identification of critical microservices is crucial for preventing invalid operations on non-critical microservice soft

Parameter	Value
Experience Replay Capacity (D)	10^4
Minibatch	64
Learning Rate	1e-3
Reward Discount (γ)	0.9
Target Model Update Frequency (C)	10
Exploration Noise Probability (ϵ)	0.1
# Hidden Layers × # Hidden neurals	2×40
Speed Up Sampling Period	100ms

Table 1: DQN Training Parameters

resource reallocation within a large-scale system [19]. Inspired by the approach in FIRM [28], we employ a two-step method for localizing the critical service. Firstly, we assess the resource utilization of each microservice. Secondly, we calculate the Congestion Ratio, which is the ratio of the 99th percentile latency to the 50th percentile latency. A high resource utilization indicates that the microservice has reached its capacity, while a high Congestion Ratio suggests that requests at the tail-end are congested within the current microservice. In both cases, it is necessary to adjust the corresponding concurrency setting.

DQN Optimal Concurrency Estimator selects the action with the highest expected reward based on the RL model and triggers the Concurrency Adapter to execute the action. Our DQN has a moderate feedback loop (i.e., the timestep is 1s), which is sufficient for action execution. A too-long control loop will prolong the convergence of DQN and incur a longer SLO mitigation time. On the other hand, a too-short control loop (e.g., 0.2s) would introduce around 10% CPU overhead due to frequent concurrency adaptation.

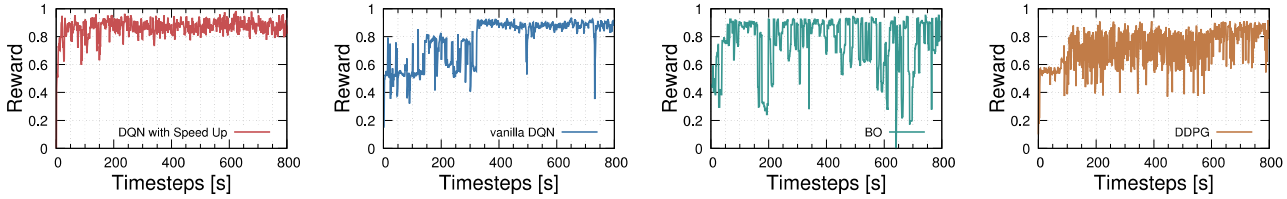
Reallocation Module includes an autoscaler to manage system hardware resource scaling and a Concurrency Adapter to execute the action command from the DQN Optimal Concurrency Estimator. Unlike previous thread adaptation frameworks (e.g., DCM [34] and ConScale [21]), our Concurrency Adapter works independently from the hardware autoscaler. This is because soft resource re-adaptation depends on the current runtime system condition through continuous monitoring, independent of hardware resource provisioning. Moreover, an optimal soft resource allocation would update when some runtime changes happen in reality, such as hardware scaling and deployment of new code versions of a microservice, requiring decoupling of RL decisions from hardware resource scaling.

4.2 Implementation Details

Request Tracing Management. Our DQN relies on distributed tracing (inspired by FIRM [28]) 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 [14]. To mitigate this problem, we first use a graph database, Neo4j, to efficiently store and query the complex invocations of services. Furthermore, we prepare a separate lightweight database (e.g., MongoDB) for each microservice to store the request timestamp records of the current service, which enables the overhead removal of the heavy filtering and aggregation tasks due to the large-scale microservices. On the other hand, we isolate the resource (e.g., CPU) for the monitoring agent and microservices in each VM to avoid interference with monitoring. Our overhead analysis shows a maximum CPU overhead of 5% of all loads when enabling metrics collection/tracing in μConAdapter. On the other hand, the fine-grained application-level metrics are calculated in dedicated machines, which does not add any additional overhead to the target runtime system.

DQN Agent Implementations. We implemented the DQN framework using PyTorch [27]. Inspired by existing RL resource management research (e.g., FIRM [28] and DeepScaling [36]), we designed the Q-network to contain two fully connected hidden layers with 40 hidden units, all using the ReLU activation function. This setup can achieve good learning efficiency and performance in our microservice benchmark applications. Adding more layers and hidden layers may slow down the training speed. Hyperparameters of the DQN model are listed in Table 1. Furthermore, the Q-network has four inputs for environment states and a restricted output space for different services. Target soft resource allocation has a rational range. For example, we can set 200 outputs for thread pool allocation in *Cart* as the enabled Tomcat in SpringBoot has a default of 200 *maxthreads*. Limiting the action space with profiling knowledge can mitigate the impact of sub-optimal actions during RL explorations.

Independent Hardware and Soft Resource Control Flow. We have implemented concurrency adaptation and hardware resource autoscaler separately to make our work easily coordinated with existing hardware-only resource management solutions. The two-model solution increases the interpretability of each model due to its simplicity. In contrast, adding soft resource adaptation actions to the existing hardware controller (e.g., FIRM) could lead to a state-action space explosion issue, which significantly increases the training overhead of the RL model. This is because soft resources usually have a large configuration space due to the heterogeneous service implementation (recall from Section 2.1), and the parameters and performance usually have a nonlinear relationship. For example, liberal allocation of threads or connections would degrade the service performance instead of improving it (see Figure 3). A unified controller might be the



(a) DQN with Speed Up reward gets stable at 50 timesteps
(b) Vanilla DQN reward gets stable at 320 timesteps
(c) BO reward does not get stable within 800 timesteps
(d) DDPG reward gets stable at 650 timesteps

Figure 7: Convergence of our DQN model with Speed Up and other dynamic online models (i.e., vanilla DQN, Bayesian Optimization (BO), and DDPG under a static workload scenario for a *Cart* service instance. The DQN with Speed Up is more effective than vanilla DQN, BO, and DDPG in online concurrency adaptation.

Execution Path	*Critical Service*	*Accuracy*
ReadHomePost	Home-Timeline	95%
	Post-Storage	90%
ReadHomeTimeline	Home-Timeline	93%
	Home-Timeline-mongodb	85%
GetFollower	Home-Timeline	92%
	Social-Graph	95%

Table 2: Accuracy of critical service localization for three execution paths in *SocialNetwork* application.

ideal solution for this joint optimization problem, assuming the training overhead can be significantly reduced.

5 EXPERIMENTAL EVALUATION

In this section, we first evaluate the accuracy of our critical service localization during a single-bottleneck test (Section 5.1). We then examine the efficiency of our proposed Speed Up mechanism in accelerating the DQN model's training process (Section 5.2). We evaluate the effectiveness of μConAdapter in assisting the hardware-only autoscaler (i.e., Kubernetes Autoscaling [16] and FIRM [28]) in stabilizing performance fluctuations under six realistic bursty workload scenarios [9] (Section 5.3). Furthermore, we compare our μConAdapter with the state-of-the-art soft resource readaptation framework, ConScale [21], when faced with various runtime system conditions changes (Section 5.4).

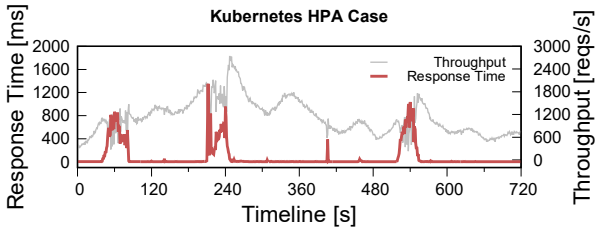
5.1 Critical Service Localization

To assess the accuracy of our critical service localization module, we tested three execution paths within the *SocialNetwork* application and configured a single bottleneck service along the path by limiting CPU resources (e.g., 1 vCPU). For each path, we tested the accuracy of two critical service estimation cases. For example, we examined Home-Timeline and Post-Storage as the critical services when composing requests "ReadHomePost". We gradually increased the workload and recorded the resource utilization and congestion ratio of each service for critical service estimation. We selected workloads uniformly at random within a range where the workload can cause the 95th percentile end-to-end latency in [0.5s, 2s]. Table 2 reveals that the average accuracy is 91.6%, supporting the effectiveness of our critical service localization module.

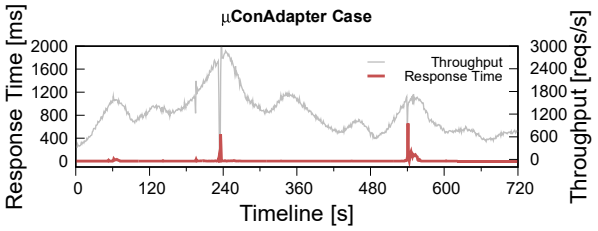
5.2 Fast Convergence of Online Concurrency Adaptation Using DQN with Speed Up

To understand the convergence behavior of our DQN model, we conducted extensive experiments using a realistic workload with a fixed number of concurrent users (i.e., 1500) to train our RL model during runtime, and the model would converge and generate the optimal resource allocation for the static system condition. We set the think time in our RUBBoS workload generator to be 1s between two consecutive requests and allocate the bottleneck service *Cart* with a 2-core CPU limit. The model will converge and recommend the optimal soft resource allocation to reach the highest reward with a stable learning process.

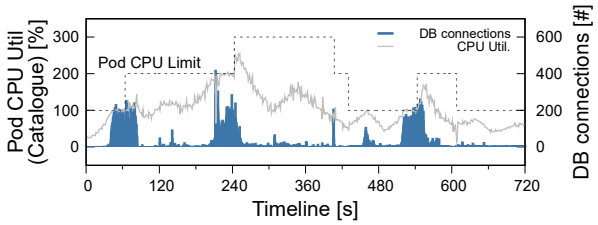
Figure 7(a) shows that the reward for our DQN model rapidly increases at the beginning of the training process and becomes stable (reaches about 0.9) at around 50 timesteps. For comparison, Figure 7(b) shows that the reward for vanilla DQN converges at around 320 timesteps, which is 6× slower than our DQN model with the Speed Up mechanism. This is because our DQN model has more sampling data for training (10×) than the vanilla DQN model due to the fine-grained sampling interval (100ms in our DQN model vs. 1s in the vanilla DQN model). Our Speed Up mechanism helps the DQN model exploit 10 virtual interactions with the runtime system within one physical timestep. These virtual interactions can provide actual knowledge of the current system state since actions and rewards are derived from runtime correlations between concurrency and performance metrics (Recall Figure 4). Hence, the DQN model with Speed Up already has 500 interactions experienced at 50 timesteps, and the vanilla DQN starts to converge until it accumulates the actual 320 interactions. However, such a benefit of sampling is not a free lunch. Due to large variations in measurement, A



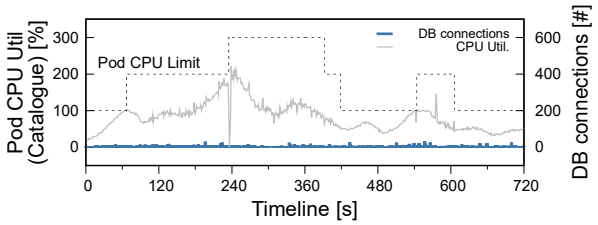
(a) The system response time spikes appear during 40s ~ 82s, 210s ~ 250s, and 520s ~ 555s under the "large variation" workload trace without μConAdapter. The system throughput drops during these periods.



(b) The system response time is more stable compared to Figure 8(a) under the same workload trace with μConAdapter and system throughput matches the workload trace.



(c) *Catalogue* scales out along with Pod CPU utilization at 62s, 241s, and 542s. Opened database connections presented spikes during the period from 40s ~ 82s, 210s ~ 250s, and 520s ~ 555s.



(d) *Catalogue* scales out along with Pod CPU utilization at 68s, 239s, and 542s. μConAdapter helps maintain a limited number of database connections during the whole period.

Figure 8: Performance comparison between Kubernetes HPA (Figures (a)(c)) and μConAdapter (Figures (b)(d)) under the same "Large Variation" workload. Our μConAdapter outperforms Kubernetes HPA in stabilizing response time fluctuations due to the DQN agent limiting the database connection pool size to around 10.

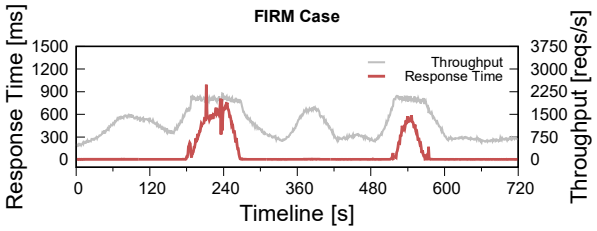
low sampling rate (e.g., 20ms) may not contribute to accurate concurrency and performance metrics. Our Speed Up mechanism boosts the process of exploration, standing orthogonal to existing exploration strategies (e.g., ϵ-greedy).

We further compare the training process of our DQN with a BO model and a DDPG model under the same static workload scenario to validate the benefits of DQN for concurrency adaptation. Figure 7(a) and Figure 7(c) show that our DQN with Speed Up can converge faster and have a more stable learning process than the BO model since BO only recommends short-term optimal actions based on prior observations. The frequent sub-optimal trials caused by system state variations would incur frequent SLO violations during the online learning process. Figure 7(d) shows that the DDPG model has a much longer training convergence time (650 timesteps), which is 13× longer than our DQN with the Speed Up mechanism. This is because DDPG involves more hyper-parameters by adopting an actor-critic approach consisting of an actor-network and a critic-network (DQN uses one network). Moreover, DDPG determines actions based on the actor-network and sometimes incurs action skew to degrade the performance. The reward curve fluctuations in Figure 7(c) and 7(d) are caused by unstable action policies. By contrast, DQN is more stable as it directly tunes the Q-value for the available concurrency settings.

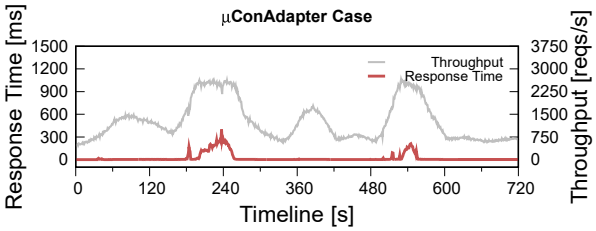
5.3 Complementing Hardware-only Autoscaling Solutions

Autoscaler Setup. We deploy μConAdapter, Kubernetes Autoscaling, and FIRM in our private cluster. The Kubernetes Autoscaling employs a rule-based scaling policy by monitoring resource utilization of microservice instances (e.g., Pod CPU utilization > 80%) and supports both horizontal (HPA) and vertical scaling (VPA). FIRM offers an RL-based fine-grained hardware resource management for microservices. We conduct evaluation experiments using three representative microservices (i.e., *Catalogue*, *Cart*, and Home-timeline services) from two benchmark applications. We configure the bursty workload that adjusts the number of concurrent users every 30 seconds by following six real-world traces [9]. Each user follows a Markov chain model to navigate the target benchmark applications with an average of 1-second think time between consecutive HTTP requests. The maximum concurrent users for *Catalogue*, *Cart*, and Home-Timeline services are 3000, 3500, and 5200, respectively, and the duration of each workload trace is 12 minutes.

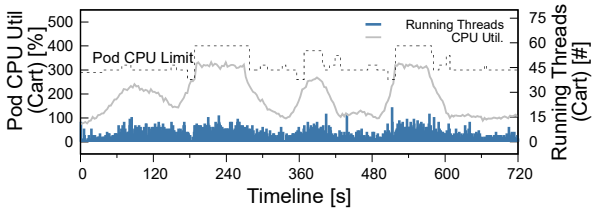
Horizontal Scaling. Figure 8 compares the system performance (i.e., throughput and response time) between Kubernetes HPA and μConAdapter under the same "Large Variation" workload trace for a *Catalogue* service. *Catalogue*



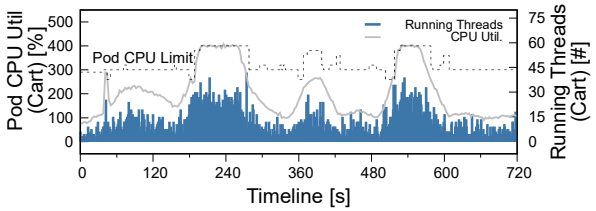
(a) The system response time spikes appear during 180s~260s, 500s~570s under the "large variation" workload.



(b) Relatively stable system response time under the same workload trace in (a) with μConAdapter.



(c) FIRM manages the CPU limit for *Cart* service. The Pod CPU resources are under-utilized after *Cart* scales up to 4-core due to the under-allocation of server threads.



(d) FIRM manages the CPU limit for *Cart* service. The Pod CPU resources can be fully utilized after *Cart* scales up to 4-core due to optimal thread pool reallocation conducted by μConAdapter.

Figure 9: Performance comparison between FIRM (Figures (a)(c)) and μConAdapter (Figures (b)(d)) under the same "Large Variation" workload. Our μConAdapter can help FIRM to stabilize response time fluctuation by re-adapting thread pool allocation to match the bursty workload.

adopts an on-demand database connection strategy by default, which establishes new connections as needed. Our μConAdapter achieves a relatively stable response time and throughput in a 12-minute experiment than that in the Kubernetes HPA case (see Figures 8(a) and 8(b)). For example, large response time fluctuations and throughput drops appear in the Kubernetes HPA case during the peak workload phases (40s~82s, 210s~250s, and 520s~555s). Taking the period 520s~555s in Figures 8(a) and 8(c) as an example, before adding the new *Catalogue* instance at 542s, we note that the number of database connections starts to accumulate and significantly affects the system performance. Once the second *Catalogue* serves new incoming requests, the downstream *Catalogue*-db would receive double concurrent requests. The high concurrent requests would further degrade the CPU efficiency of the database service (i.e., 542s~555s). On the other hand, our μConAdapter can stabilize response time and throughput during the whole experiment runtime, as shown in Figure 8(b). This is because μConAdapter limits the database connection pool size to 10 based on our DQN model (see Figure 8(d)), which helps avoid large response time spikes during the temporary overloading.

Vertical Scaling. We then validate the effectiveness of our μConAdapter when adopting vertical scaling for stateful microservices (e.g., *Cart*). Unlike horizontal scaling, vertical scaling (i.e., adding or removing vCPU) would mitigate the impact of the complex dependencies between upstream and downstream microservices. However, it still requires concurrency adaptation to avoid performance degradation due to the lack of coordination between hardware and soft resources. We initially set *Cart* thread pool to be 6, which is optimal for *Cart* with 2-core vCPU through pre-profiling.

We compare our μConAdapter with both the Kubernetes VPA and FIRM [28]. Figure 9 shows the comparison between FIRM and μConAdapter under the same "Large Variation" workload trace for a *Cart* service. Figure 9(c) shows that FIRM can tune CPU limit with fine granularity and provide faster scaling to help reduce SLO violations. However, it still cannot avoid the large response time spikes during the temporary overload phase (180s~260s and 500s~570s) in Figure 9(a). A response time spike appears even though FIRM scales up the CPU limit to 4-core (maximum allocation) from 180s to 260s (see Figures 9(a) and 9(c)). FIRM did not adapt the thread pool accordingly after scaling hardware resources, making the original optimal threads allocation insufficient to fully utilize the hardware resources (e.g., CPU) as we studied in Section 2.3. The CPU utilization of *Cart* is about 310% even though the CPU limit is scaled up to 4-core, leading to *Cart* CPU's low efficiency and sub-optimal system performance. On the other hand, our μConAdapter dynamically adapts *Cart* thread pool from 6 to 25 after several trials to match the updated vCPU allocations in Figure 9(d).

We further compare the average throughput and tail latency (i.e., 95th and 99th percentile) between the hardware-only scaling frameworks and our μConAdapter under other types of workload traces in Table 1. Our μConAdapter can

Workload Trace	Cart service in Sock Shop			Home Timeline service in Social Network		
	95th Percentile Response Time [ms]	99th Percentile Response Time [ms]	Throughput [reqs/s]	95th Percentile Response Time [ms]	99th Percentile Response Time [ms]	Throughput [reqs/s]
	K8s VPA/FIRM/**μConAdapter**	K8s VPA/FIRM/**μConAdapter**	K8s VPA/FIRM/ **μConAdapter**	K8s HPA/ **μConAdapter**	K8s HPA/ **μConAdapter**	K8s HPA/ **μConAdapter**
Large Variation 	659 / 501 / **203**	768 / 592 / **315**	1142 / 1185 / **1261**	499 / **189**	1027 / **354**	1673 / **1938**
Quick Varying 	726 / 500 / **214**	808 / 553 / **303**	1651 / 1732 / **1990**	469 / **185**	860 / **305**	2082 / **2579**
Slowly Varying 	704 / 663 / **326**	748 / 749 / **472**	1078 / 1196 / **1246**	461 / **164**	1047 / **414**	1809 / **2050**
Big Spike 	498 / 535 / **186**	600 / 642 / **297**	702 / 731 / **741**	520 / **219**	1171 / **403**	1230 / **1438**
Dual Phase 	687 / 551 / **268**	753 / 633 / **358**	1421 / 1472 / **1699**	314 / **84**	1122 / **334**	2120 / **2334**
Steep Tri Phase 	744 / 624 / **288**	803 / 687 / **432**	1283 / 1318 / **1466**	283 / **68**	1045 / **188**	1481 / **1705**

Table 3: Tail response time (i.e., 95th and 99th percentile) and throughput comparison between Kubernetes HPA, Kubernetes VPA, FIRM, and μConAdapter under six realistic bursty workload traces for two representative services from *SockShop* and *SocialNetwork*.

significantly reduce the 95th and 99th percentile latency by 2× than Kubernetes VPA and FIRM. We further evaluate the effectiveness of μConAdapter in a more sophisticated benchmark, SocialNetwork. Our μConAdpater achieves 3× lower on average 95th and 99th percentile latency for the Home-Timeline service than Kubernetes HPA.

5.4 Performance Comparison between μConAdapter and ConScale

ConScale [21] is a state-of-the-art framework that coordinates fast concurrency adaptation with autoscaling to stabilize the system response time, which performs similarly to μConAdapter when optimal concurrency remains stable due to little runtime changes (e.g., vertical scaling and microservice rolling updates). We argue that our μConAdapter shows better adaptability due to the RL model compared to ConScale's online statistical SCT model. Our μConAdapter adjusts soft resources every second, while ConScale reallocates soft resources only after hardware resource scaling. Moreover, the SCT model needs a coarse 3-minute time window to reconstruct the correlation model and recommend a new optimal setting. Such a reaction window highly affects the quality of model estimation. This is because ConScale employs online regression using a batch of recent observations. A short reaction window with insufficient new data would degrade ConScale's decision. Figure 10 validates that μConAdapter can provide better concurrency adapting under the same "Large Variation" workload trace facing the system condition changes (e.g., upgrading of microservices business logic and drifting of system state).

Upgrading of Microservices Business Logic. We start our experiments with the original *Catalogue* service, then upgrade the code to an optimized version with fewer computations for organizing the socks, which reduces the service time of the corresponding requests. Figure 10 shows that μConAdapter and ConScale have comparable response time performance before *Catalogue* version upgrade at 401s since both frameworks adopt optimal database connections. However, μConAdapter outperforms ConScale in achieving a much more stable low response time during the period 540s∼570s in Figure 10(a). This is because ConScale updates the new optimal database connections at 581s, and *Catalogue* scales out at 565s in Figure 10(b)). The delay of database connection reallocation causes a response time spike (period 540s∼565s) since the previous optimal setting for the original *Catalogue* is no longer valid (Figure 10(c)).

We apply Pareto analysis [18] to evaluate whether μConAdapter does optimal soft resource allocation. A Pareto optimal solution to a multi-objective optimization problem should be equally good or better in all objective functions (at least one). We apply a Pareto analysis on the performance/cost tradeoffs of μConAdapter after microservice updates (i.e., 401s∼701s) in Figure 10. We first extract the performance reward of each action of μConAdapter every second. Then we measure the sum of normalized CPU and memory utilization as the resource cost. Figure 11(a) shows that μConAdapter's database connections adaptation follows the Pareto frontier (i.e., 0.14 deviation from the Pareto front of every decision), and resource cost and performance reward are linearly dependent, indicating that μConAdapter helps *Catalogue* fully utilize the hardware resources and achieve optimal performance. By contrast, ConScale's suboptimal actions cause a low-performance reward (e.g., < 0.8) while incurring high resource cost (e.g., > 1.5). This is far from the Pareto frontier and also exhibits a larger deviation of 0.213 (see Figure 11(b)).

Drifting of System State. In this set of experiments, we initially launch *Cart-db* service with the original dataset and then manually enlarge the dataset to compare the

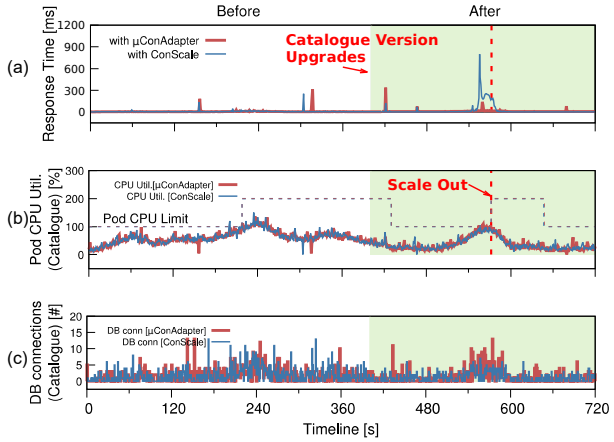
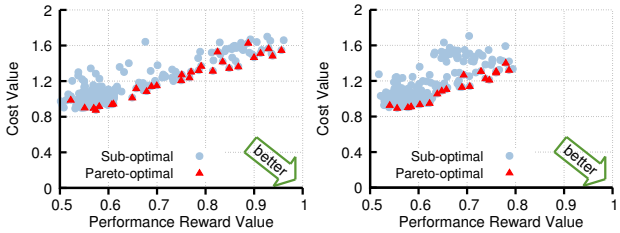


Figure 10: μConAdapter achieves more stable response time than that in ConScale case after *Catalogue* service upgrades to use fewer computations (i.e., 401s~720s).



(a) μConAdapter follows the Pareto frontier and fully utilizes the critical resources.

(b) ConScale exploits sub-optimal actions that cause high resource cost and a low-performance reward.

Figure 11: Pareto analysis on performance/cost trade-offs of μConAdapter and ConScale from 401s~701s in Figure 10. μConAdapter does Pareto-optimal soft resources allocation while ConScale suffers from resource inefficiency due to sub-optimal allocations.

performance differences between μConAdapter and ConScale. Figure 12 shows a consistent experimental result that μConAdapter outperforms ConScale in stabilizing response time spikes after *Cart-db* database update (i.e., enlarging *Cart-db* dataset) at 401s. ConScale has encountered higher response time spikes than those in the μConAdapter case during the period 500s~600s in Figure 12(a). ConScale fails to adjust the thread pool since no hardware resource scaling was triggered after *Cart-db* updates and previous optimal server threads cannot fully utilize all 4-core CPU resources (e.g., about 340%). In contrast, μConAdapter captures fine-grained system state data with a tight feedback loop and quickly generates new server thread recommendations. Therefore, our experimental results demonstrate our proposed μConAdapter can effectively provide better adaptivity and reliability to fast and accurately adapt the optimal soft resource allocation to various system condition changes.

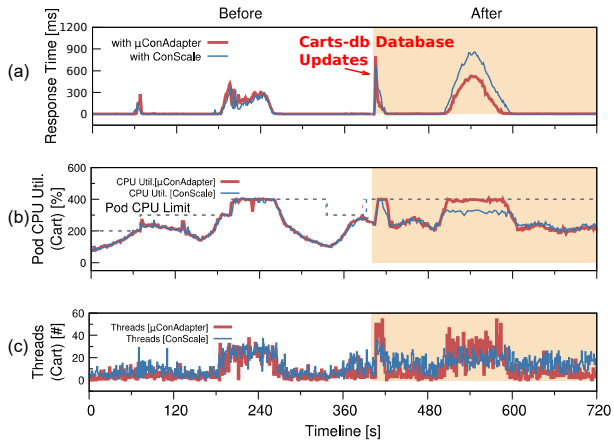


Figure 12: Our μConAdapter achieves more stable response time than that in ConScale case after *Cart-db* service upgrades to persist an enlarged dataset (i.e., 401s~720s).

5.5 Runtime Overhead Analysis

Overhead Imposed by RL Update. We used PyTorch to implement a DQN-based model for optimal concurrency adaptation. Initially, we fine-tuned each microservice's DQN model by considering a specific hardware resource allocation and learning from diverse system states resulting from bursty workloads. Take the *Cart* service as an example. In the initialization phase, thanks to the small number of hyper-parameters and our Speed Up mechanism, our DQN model took just 50 timesteps to identify the optimal concurrency. Meanwhile, the minimal cost of soft resource adaptation operations ensures tight feedback from RL agents. For example, thread pool adaptation in the *Cart* service required an average latency of 62ms in all workloads, which is considered an acceptable delay in practice. Moreover, the frequency of RL model updates depends on the runtime changes in the system/workload. Since RL online training operates through trial and error, it can produce sub-optimal decisions, particularly during the initial stages of workload changes.

Metrics Collection/Tracing Overhead. Similar to FIRM, we utilize the distributed tracing method to record the arrival and departure timestamps of individual requests within the target microservice at millisecond granularity. Enabling metrics collection/tracing in μConAdapter results in a maximum of 5% CPU overhead under all workloads. Subsequently, we generate fine-grained application-level metrics for the critical service by processing the arrival and departure timestamps of each request it serves, which are logged in every component service. The overhead of generating the required application-level metrics at a 100ms sampling interval is recorded in Table 4, which were obtained from a dedicated

server equipped with 8 cores of Intel Xeon E5-2603v3 processors and 8GB of RAM in the SockShop application. Consequently, the collection of application-level metrics has a negligible impact on the runtime system of the target.

6 LIMITATIONS

μConAdapter has several limitations that we plan to address in future work. Firstly, μConAdapter currently focuses on two generic soft resources (i.e., server threads and connection) adaptations among most mid-tier and database services. More types of soft resources (e.g., heap region size for memory management) and services (e.g., Memcache) are still under investigation. Secondly, μConAdapter is designed for closed-loop workloads that emulate the behaviors of concurrent users for web applications. Evaluating μConAdapter's performance with open-loop workloads can improve the generality of μConAdapter for modern cloud systems. Thirdly, we plan to refine μConAdapter's input state design to ensure precise detection of runtime changes and model retraining. A recent study [29] inspires us to consider using meta-learning to facilitate model adaptation in dynamic environments.

7 RELATED WORK

Autoscaling frameworks for microservices mainly focus on elasticizing computing resources (e.g., CPU and memory) in clouds [11, 13, 22, 25, 28, 30, 38] For example, Autopilot [30] uses machine learning algorithms applied to historical data about the prior execution of a job to predict the CPU/memory usage of jobs. FIRM [28] leverages fine-grained measurement data and machine-learning methods to fast and dynamically provision hardware resources to mitigate SLO violations caused by low-level resource contention. However, these approaches barely discuss the scaling of soft resources (e.g., threads or connections) that control the concurrent use of hardware resources, which could become significant sources of performance fluctuations (see Section 2.3).

Critical service localization in microservices is challenging, and many works have studied critical path analysis from different perspectives [14, 23, 28, 41]. For example, Kaldor et al. [14] track requests from web browsers/mobile to backend services by developing an end-to-end tracing system, Canopy, which can handle billions of traces. Liu et al. [23] detect the performance anomaly using a Deep Bayesian Network in an unsupervised manner. Zhang et al. [41] use critical path analysis over RPC traces to bubble up interesting activities and discard noisy events. These works provide insights into utilizing distributed request traces for automated anomaly detection, which provides a good foundation for critical service localization.

Software reconfiguration to mitigate SLO violations for cloud applications has been studied extensively before [5, 20, 21, 32, 35]. For example, Sriraman et al. [32] develop μTune, which automatically chooses load-optimal threading models for microservices based on various offered loads to improve tail latency. Zhang et al. [40] leverage machine learning models to automatically recommend new configurations for database management systems to improve the system performance. ConScale [21] and Sora [20] adopt statistical correlation models to quickly estimate the optimal soft resource configurations of key servers during the system scaling process. Our work complements their work by integrating a reinforcement learning-based model, capable of capturing each subtle change in system conditions and better adapting soft resource allocations for microservices.

Num Requests	Request Size	Processing Time	Memory Usage
10K	0.5MB	0.694 sec	49MB
20K	1.1MB	0.971 sec	65MB
40K	2.1MB	1.291 sec	81MB
80K	4.2MB	1.941 sec	90MB

Table 4: Overhead of generating application-level metrics at 100ms sampling interval in *SockShop*.

8 CONCLUSION

We present μConAdapter, a framework for optimal concurrency adaptation that integrates fast and dynamic soft resource reallocation for critical microservices with existing hardware-only autoscalers. Our experiments, conducted on two representative microservices benchmarks (SockShop and SocialNetwork) using six realistic bursty workload traces, demonstrate the effectiveness of μConAdapter in reducing tail latency at the 99th percentile compared to hardware-only scaling strategies like Kubernetes Autoscaling and FIRM. Additionally, μConAdapter outperforms the concurrency adaptation framework, ConScale, facing microservices updates. Overall, μConAdapter enables swift mitigation of system response time fluctuations by combining efficient hardware and soft resource provisioning. It contributes to high resource efficiency and optimal performance in meeting the demands of modern cloud applications.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd Dr. Timothy Zhu for their valuable comments that improved this paper. This research has been partially funded by the National Science Foundation by CNS (2000681), CNS (2245827), 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] Microservice architecture diagram examples. https://www.devteam.space/blog/microservice-architecture-examples-and-diagram/.

[3] Tony mauro. adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[4] Sock shop microservice demo application. https://microservices-demo.github.io/, 2016.

[5] Banerjee, S. S., Jha, S., Kalbarczyk, Z., and Iyer, R. K. Bayesperf: minimizing performance monitoring errors using bayesian statistics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 832–844.

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

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

[8] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., and Delimitrou, C. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (April 2019).

[9] 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.

[10] Gou, S. Z., and Liu, Y. Dqn with model-based exploration: efficient learning on environments with sparse rewards. *arXiv preprint arXiv:1903.09295* (2019).

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Klinaku, F., Frank, M., and Becker, S. Caus: An elasticity controller for a containerized microservice. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (2018), pp. 93–98.

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

[17] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[18] Li, Q., Li, B., Mercati, P., Illikkal, R., Tai, C., Kishinevsky, M., and Kozyrakis, C. Rambo: Resource allocation for microservices using bayesian optimization. *IEEE Computer Architecture Letters 20*, 1 (2021), 46–49.

[19] 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.

[20] Liu, J., Wang, Q., Zhang, S., Hu, L., and Da Silva, D. Sora: A latency sensitive approach for microservice soft resource adaptation. In *Proceedings of the 24th ACM/IFIP/USENIX Middleware Conference (Middleware)* (2023).

[21] 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.

[22] Liu, N., Li, Z., Xu, J., Xu, Z., Lin, S., Qiu, Q., Tang, J., and Wang, Y. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *2017 IEEE 37th international conference on distributed computing systems (ICDCS)* (2017), IEEE, pp. 372–382.

[23] Liu, P., Xu, H., Ouyang, Q., Jiao, R., Chen, Z., Zhang, S., Yang, J., Mo, L., Zeng, J., Xue, W., et al. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)* (2020), IEEE, pp. 48–58.

[24] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature 518*, 7540 (2015), 529–533.

[25] 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.

[26] 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.

[27] PyTorch. Pytorch official website. https://pytorch.org/.

[28] 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.

[29] Qiu, H., Mao, W., Wang, C., Franke, H., Youssef, A., Kalbarczyk, Z. T., Başar, T., and Iyer, R. K. {AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (2023), pp. 387–402.

[30] Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–16.

[31] Sachidananda, V., and Sivaraman, A. Learned autoscaling for cloud microservices with multi-armed bandits. *arXiv preprint arXiv:2112.14845* (2021).

[32] 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.

[33] Tan, T., Chu, T., and Wang, J. Multi-agent bootstrapped deep q-network for large-scale traffic signal control. In *2020 IEEE Conference on Control Technology and Applications (CCTA)* (2020), IEEE, pp. 358–365.

[34] 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.

[35] Wang, S., Li, C., Hoffmann, H., Lu, S., Sentosa, W., and Kistijantoro, A. I. Understanding and auto-adjusting performance-sensitive configurations. *ACM SIGPLAN Notices 53*, 2 (2018), 154–168.

[36] Wang, Z., Zhu, S., Li, J., Jiang, W., Ramakrishnan, 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* (2022), pp. 16–30.

[37] 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.

[38] 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.

[39] Yu, G., Chen, P., and Zheng, Z. Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. *IEEE Transactions on Cloud Computing* (2020).

[40] 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.

[41] Zhang, Z., Ramanathan, M. K., Raj, P., Parwal, A., Sherwood, T., and Chabbi, M. {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022), pp. 655–672.