Practical Efficient Microservice Autoscaling with QoS Assurance

Md Rajib Hossen
The University of Texas at Arlington
Texas, USA
mdrajib.hossen@mavs.uta.edu

Mohammad A. Islam
The University of Texas at Arlington
Texas, USA
mislam@uta.edu

Kishwar Ahmed
University of South Carolina Beaufort
South Carolina, USA
ahmedk@uscb.edu

ABSTRACT

Cloud applications are increasingly moving away from monolithic services to agile microservices-based deployments. However, efficient resource management for microservices poses a significant hurdle due to the sheer number of loosely coupled and interacting components. The interdependencies between various microservices make existing cloud resource autoscaling techniques ineffective. Meanwhile, machine learning (ML) based approaches that try to capture the complex relationships in microservices require extensive training data and cause intentional SLO violations. Moreover, these ML-heavy approaches are slow in adapting to dynamically changing microservice operating environments. In this paper, we propose PEMA (**P**ractical **E**fficient **M**icroservice **A**utoscaling), a lightweight microservice resource manager that finds efficient resource allocation through opportunistic resource reduction. PEMA's lightweight design enables novel workload-aware and adaptive resource management. Using three prototype microservice implementations, we show that PEMA can find efficient resource allocation and save up to 33% resource compared to the commercial rule-based resource allocations.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**.

KEYWORDS

Autoscaling, microservices, resource management, cloud computing, quality of service

ACM Reference Format:
Md Rajib Hossen, Mohammad A. Islam, and Kishwar Ahmed. 2022. Practical Efficient Microservice Autoscaling with QoS Assurance. In *Proceedings of the 31st Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3502181.3531460

1 INTRODUCTION

Motivation. Microservices architecture is enjoying a growing penetration in user-facing cloud applications where an ensemble of loosely-coupled and small service components (i.e., microservices) work together to serve user requests [1–3]. As illustrated in Fig. 1, microservices architecture is a significant departure from traditional monolithic deployments with a few large application layers

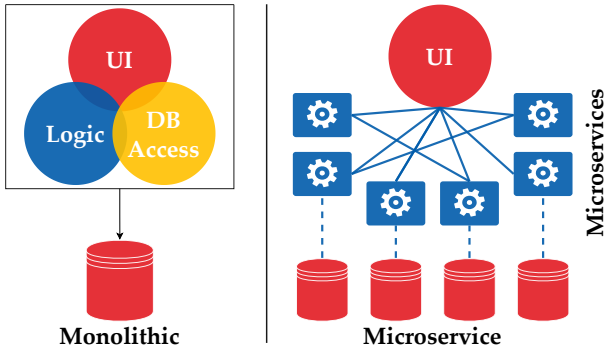


Figure 1: Comparison between monolithic and microservices architecture. Microservices consists of many small loosely coupled systems.

such as user-facing front-end, back-end business logic, and database [4]. Unlike monolithic applications, the small microservices can be easily managed and kept updated by small dedicated DevOps teams [5]. Moreover, microservices are typically stateless and communicate using lightweight APIs [6, 7]. Hence, they offer agile resource management and scaling, better fault tolerance, and great platform agnostic compatibility among different microservices that cannot be matched by monolithic applications [5, 8].

Microservices come with their own sets of challenges, and in this paper, we focus on its resource management. In principle, microservice resource management is same as monolithic applications - achieve the desired performance (e.g., end-to-end response latency) with the minimum resource allocation [9–11]. Resource management for microservices-based applications, however, is more challenging because these applications have a much larger configuration space due to the sheer number of microservices responsible for the application performance. For example, if we consider an application with m microservices where each microservice can be configured with n different CPU allocations, there will be n^m possible resource configurations. Moreover, microservices have complex communication topology and inter-dependencies that make it harder to identify and mitigate Quality of Service (QoS) violations [2, 8]. A single user request may traverse through several microservices, and if any microservice in the critical path becomes a bottleneck, the end-to-end response time will increase significantly [12]. Our motivating experiments on three prototype microservices show that the same amount of CPU allocation can result in more than 250% increase in application latency based on how the resource is distributed among different microservices. Meanwhile, existing resource management techniques developed for monolithic applications with a few service layers cannot readily capture the complex microservice interactions to make effective resource allocation choices [13–16].

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9199-3/22/06.
https://doi.org/10.1145/3502181.3531460

Nevertheless, addressing these resource management challenges for microservices is of paramount importance as an increasing number of production cloud services have been adopting microservice architectures [4, 17].

Limitation of state-of-art approaches. Owing to the growing interest, several recent works try to address the resource management challenges in microservices [4, 12, 17–19]. They focus on utilizing machine learning (ML) techniques to capture the complex relationship between microservice resources and performance. For instance, FIRM [12] uses a combination of support vector machines (SVM) and reinforcement learning to localize root causes of SLO violations, and apply resource autoscaling to avert these violations. Sage [18], on the other hand, uses supervised training to identify dependencies between different microservices using a Causal Bayesian Network, and a graph encoder to track the QoS violating microservices to adjust their resources. However, this line of works built on ML are fundamentally limited by their extensive training requirements, both in terms of training time to capture the dynamics of the microservices and data resolution (e.g., request level traces to build dependency graphs). More importantly, to learn from the data, some ML-based techniques intentionally cause or allow SLO violations which is undesirable in production systems [12, 17–19]. Also, any changes in the microservices architecture and inter-dependencies will require retraining the system. This ML retraining can become a barrier for real world microservices applications which go through frequent software/code updates. ML retraining can also be triggered by changes in underlying cloud hardware due to server migrations and upgrades. On the other hand, the resource demand of microservices changes with the workload on a daily basis. However, existing approaches focusing on SLO violations do not directly incorporate dynamic workload in their learning [12, 17–19].

Key insights and contributions. To avoid the hurdles of the approaches mentioned above, we propose PEMA (**P**ractical **E**fficient **M**icroservice **A**utoscaling), a lightweight microservice resource manager that does not need extensive training. PEMA utilizes iterative feedback-based tuning to find efficient resource allocations that satisfy the SLO. Instead of finding the best resource configuration, PEMA first allocates abundant resources to all microservices to satisfy SLO and then tries to exploit resource reduction opportunities. Allocating abundant resources for the microservices can be easily accomplished as cloud native applications enjoy a great degree of resource scalability. The initial (and inefficient) resource allocation can be achieved using existing rule-based resource managers [20]. Using this opportunistic resource reduction approach, PEMA avoids causing intentional SLO violations as it always allocates enough resources for microservices, even when performing poorly (i.e., missing resource reduction opportunities). To enable PEMA's approach, we introduce the notion of "monotonic resource reduction" where we either reduce the resource of a microservice or keep it unchanged. In contrast, a non-monotonic resource reduction can be made through resource reduction for some microservices and resource increase for some other microservices with an overall total resource reduction (i.e., a greater total reduction than total increase). We observe that monotonic resource reductions result in a monotonic increase in response time. Hence, we can use the response time as feedback to identify resource reduction opportunities to make gradual monotonic resource changes to reach efficient allocations. In addition, based on experiments on prototype microservice implementations, we identify that we can avoid resource reduction in bottleneck services using only two microservice-level performance metrics - CPU utilization and CPU throttling time.

Our feedback-based design also allows us to seamlessly adapt a workload-aware design where we implement a novel approach of using dynamic workload ranges with a dynamic response time target. More specifically, to avoid time-consuming learning of the efficient allocation for different workload levels independently, we use dynamic ranging where PEMA starts resource allocation for a large workload range (e.g., 100~1000 requests-per-second) and then gradually splits them into smaller ranges (e.g., 100~200 requests-per-second). We retain the resource allocation learned by the parent workload range during the range split to bootstrap the tuning for the new workload range. Based on the workload, we also dynamically alter the feedback from response time to allow headroom for response time change due to workload change.

Our performance evaluation reveals that PEMA can attain resource efficiency close to the optimum [1] with high probability. We also show that PEMA can save as much as 33% resource compared to rule-based resource allocation strategies of commercially available cluster managers. We demonstrate that PEMA can seamlessly adapt to changes in microservice deployment due to changes in underlying cloud hardware. Moreover, we show that adaptability of PEMA allows its integration with opportunistic resource management where variable SLO is used for trading performance for resource savings.

Experimental methodology and artifact availability. We use three prototype microservices implementations widely used in academic research on microservices [12, 17, 18]. We implement `TrainTicket` from [21] consisting of 41 microservices, `SockShop` from [22] with 13 microservices, and `HotelReservation` from [2] with 18 microservices. We deploy these services in Docker [23] containers managed by Kubernetes [24]. Our Kubernetes cluster consists of five nodes with one master node and four worker nodes. Each node is equipped with two 10-core Intel Xeon processors, and 128 GB of Memory running the Ubuntu 20.04.3 operating system. Our software artifacts are available at our GitHub repository [25].

Limitations of the proposed approach. We share our insight on the limitations of PEMA on two different fronts - the fundamental limitations in PEMA's design approach and the limitations of PEMA's current implementation. Due to its non ML-heavy approaches, PEMA's design loses on capturing complex interdependencies between microservices, and therefore is limited on the absolute best resource efficiency it can achieve. However, PEMA makes up for this loss of optimization potential through its simplicity and adaptability to change (e.g., workload variation). Also, due to our randomized exploration based search, PEMA offers provably efficient management and can result in arbitrarily inefficient resource allocations at times. We defer the discussion on the limitations of PEMA's current implementation to the end of our paper in Section 6 to make it more meaningful to the reader.

[1] Optimum resource allocation refers to the minimum resource required to satisfy SLO. We describe how we identified the optimum resource allocation in Section 4.2.

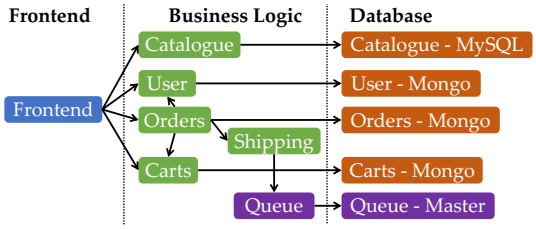


Figure 2: Architecture of the SockShop [22].

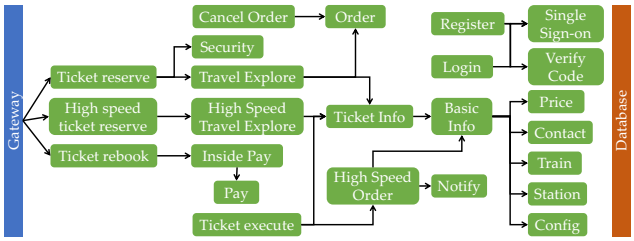


Figure 3: Architecture of the TrainTicket [21, 26].

2 PRELIMINARIES

2.1 Microservice Prototypes

SockShop [22]. SockShop implements the user-facing microservices of an e-commerce website. SockShop's functionalities include searching, order placement, and shipping. Its functionalities can be divided into three parts - front-end, business-logic, and databases. The user requests arriving at the front-end are routed to appropriate microservices to serve the requests. The business-logic interact with each other and the databases as needed. The front-end is implemented using NodeJS, orders and carts microservices are implemented using Java, and the rest of the services are implemented with Go. Shipping service uses RabbitMQ to propagate messages to Queue-Master which is implemented in Java. The databases are implemented using MySQL and MongoDB. For SockShop, we set the SLO response time to 250 milliseconds. The overall architecture is shown in Fig.2.

TrainTicket [21]. TrainTicket implements a complete train ticket booking system consisting of 41 microservices. Its functionalities include ticket search with date and destination filtering, seat booking, ordering food, payment, and consignment service. The business logic of TrainTicket is implemented using 24 microservices divided into five layers where the microservices in the upper layers depend on the microservices of the lower layers. There are some intra-layer communications as well. The overall architecture is shown in Fig. 3. TrainTicket covers many features of microservices such as synchronous invocations, asynchronous invocations, and message queues. The TrainTicket business logics and front-end are built using NodeJS, Java, Python, and Go. The databases are implemented using MongoDB, and MySQL. For TrainTicket, we set the SLO response time to 900 milliseconds.

HotelReservation [2]. HotelReservation application is adopted from DeathStarBench microservices benchmark applications. It has 18 microservices. HotelReservation lets users get nearby hotel information and reserve rooms. All the services in HotelReservation are written in Go, and they communicate with each other via gRPC

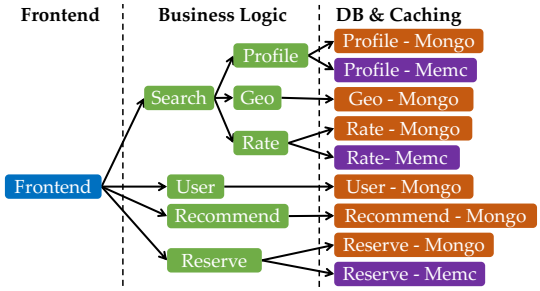


Figure 4: Architecture of the HotelReservation [2].

[27]. The back-end uses Memcached for in-memory caching to provide faster searches while the persistent databases are implemented using MongoDB. The application is pre-populated with 80 hotels and 500 registered users. This application consists of 18 microservices. For HotelReservation, we set the SLO response time to 50 milliseconds.

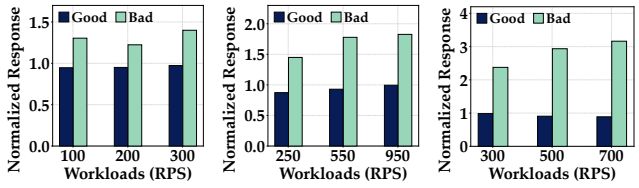
2.2 Performance Monitoring and Resource Allocation

For performance monitoring of our container-based microservice implementation, we use Prometheus [28] to collect container-specific metrics such as CPU utilization and CPU throttling. For collecting end-to-end latency performance and workload (i.e., requests per second), we use Linkerd [29]. We also use Jaeger [30] which provides detailed tracing of each request showing its service path through different microservices. Note that, our resource manager does not utilize Jaeger.

We use the 95-th percentile end-to-end response latency as a performance metric and refer to it as the application performance unless specified otherwise. For our cloud-based microservice applications which exploit request-level-parallelism, end-to-end response latency is the popular choice of performance metric [31]. For microservice resources, we only consider the total CPU allocation to a microservice with the assumption that the memory is not a bottleneck resource. Furthermore, we do not explicitly address the number of container replicas and consider homogeneous settings for each microservice.

2.3 Challenges in Microservice Resource Management

As in any general computing system, the performance of microservices applications depends on their resource allocation. Various theoretical and practical tools have been developed over the years to establish a mathematical relationship between computing resource and performance [9]. However, they are not equipped to capture complex interactions between different microservices. Any request's end-to-end response time (i.e., performance) is the aggregation, often non-linearly due to parallel processing, of time spent in many microservices. Consequently, the presence of any microservice with a resource bottleneck on the service path affects the end-to-end response time. Meanwhile, the resource demand for different microservices can be widely different based on their service. Hence, the distribution of resources among different microservices plays a crucial role in application performance.



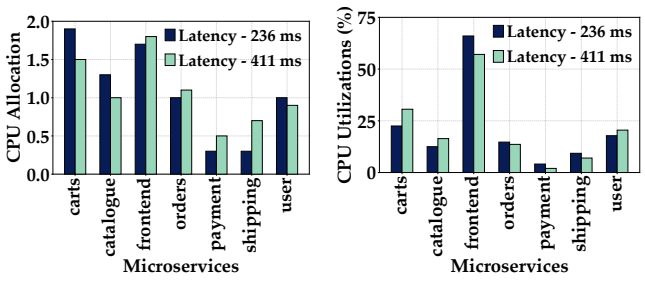
(a) TrainTicket (b) SockShop (c) HotelReservation

Figure 5: Impact of "Good" (i.e., satisfies SLO) and "Bad" (i.e., violates SLO) resource distribution on the response time normalized to the SLO at different workloads levels. In Fig. (a), for workloads 100, 200, and 300, total CPU allocations are 40.5, 42, and 47 respectively. In Fig. (b), for workloads 250, 550, and 950, total CPU allocations are 6.3, 7.7, and 14.1, respectively. In Fig. (c), for workloads 300, 500, and 700, total CPU allocations are 5.1, 6.9, and 9.4, respectively.

To demonstrate the importance of resource distribution, we run a few experiments on our microservices prototypes. We first identify "good" resource allocations that satisfy the SLOs for the prototypes for different workload levels. We then change these to "bad" distributions by randomly altering resource allocations while keeping the total resource the same. Fig. 5 shows the impact of this resource distribution - even with the same amount of resources, the performance varies significantly because of changes in distribution. For, `TrainTicket` we see as much as 43.88% increase in response time while `SockShop` and `HotelReservation` suffer up to 91.3% and 256.2% increase, respectively.

Due to the large configuration space, the "good" resource distribution cannot be readily determined for microservices. Also, the nature of processing done in different microservices is different and cannot adhere to any general resource allocation principle, such as keeping utilization lower than a certain level [10, 11, 20, 32]. To illustrate this, we show the resource distributions of `SockShop`'s microservices for the good and bad configurations with the same amount of total resource in Fig. 6(a) and the corresponding CPU utilization in Fig. 6(b). We see that there is no readily identifiable root cause (e.g., microservice with bottleneck resource) in response latency in Fig. 6(b) for the 74% increase (236 milliseconds to 411 milliseconds). Also, while we see an increase in utilization for the `cart`, `catalogue`, and `user` services for the bad configuration, their utilization remains below the `frontend`'s utilization, making it impossible to employ any common utilization-based resource allocation policy. Furthermore, we see that the utilization change due to resource change is different for different services. For example, the `frontend`'s utilization changed more than `orders` even though they experienced similar resource change. This indicates that resource allocation policies that try to increase overall utilization [12], may not be the most efficient.

To summarize, *for efficient microservice management, it is crucial to identify how resources should be distributed among different microservices as the same amount of resources can result in significantly different performance based on which microservice gets how much resources. However, finding the efficient resource distribution is very hard as there are no easily generalizable markers (e.g., high utilization) to assist in the resource allocation.*



(a) CPU allocation (b) Utilization

Figure 6: (a) Total CPU allocation of 7.5 distributed among different microservices of `SockShop`. (b) CPU utilization.

3 DESIGN OF PEMA

We have two design goals for our resource manager - (1) assure QoS (i.e., avoid SLO violations), and (2) find efficient resource allocation. Using a discrete-time model with a time step Δt (e.g., one minute) where the microservice resource allocation decisions are updated at the beginning of each time step, we formalize our resource management as the following optimization problem ORA (**O**ptimum **R**esource **A**llocation)

$$\text{ORA}: \underset{\mathbf{x}^t}{\text{minimize}} \quad \sum_{i=1}^{N} x_i^t \tag{1}$$

$$\text{subject to} \quad \mathcal{F}(\mathbf{x}^t) \leq R \tag{2}$$

Here, at time step t, $\mathbf{x}^t = (x_1^t, x_2^t, \cdots, x_N^t)$ is the resource allocation vector of the N microservices, $\mathcal{F}(\mathbf{x}^t)$ is the end-to-end latency response of the application for resource allocation $\mathbf{x}^t$, and R is the response latency threshold defined in the SLO. In what follows, we develop PEMA (**P**ractical **E**fficient **M**icroservice **A**utoscaling) - a practical microservices resource manager that finds a provably efficient solution to ORA. We first discuss the design principles of PEMA to achieve our goals (i.e., the solution to ORA), followed by the rationale for our choices and implementation details of PEMA.

Note here that, instead of minimizing the total resource allocation, ORA can also adopt cost minimization as its goal by replacing x_i^t in Eqn. (1) with $C(x_i^t)$ which represents the cost of resource x_i^t. Moreover, resource allocation vector $\mathbf{x}^t$ is not restricted to CPU allocations only. We can incorporate other types of cloud resources such as memory allocation and I/O bandwidth in $\mathbf{x}^t$. Nevertheless, our general solution principle still applies, albeit the opportunistic resource reductions need to be conducted on multiple resource dimensions.

3.1 Design Principles of PEMA

A learning-based approach. Achieving either of our design goals for a microservice-based application is non-trivial due to their complex topology and inter-dependency between different microservices. Moreover, the relation and interaction with each other for these microservices varies with applications and deployments, even among different versions of the same application. Not to mention, the underlying cloud hardware (e.g., processor type/model) hosting

these applications also affects the microservice performance and resource allocation. Consequently, our resource manager needs to identify resource allocation strategies for each microservice implementation and at the same time be able to adapt as the application evolves. Hence, we take a learning-based approach where PEMA iteratively interacts with the application through a feedback loop to navigate towards efficient resource allocations.

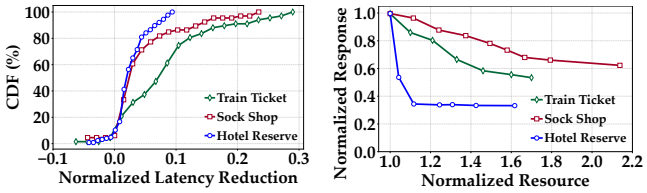
Provably efficient resource allocation. Solving PEMA can be interpreted as tuning the application resources that will make the response latency exactly equal to the SLO specified level. However, since the resource distribution across different microservices affects the latency and microservice-based applications usually consist of many microservices, there could be many different resource allocations that result in a latency equal to the SLO. Consequently, in PEMA, instead of finding the best resource allocation (i.e., the lowest aggregate resource), our goal is to find a resource allocation close to the optimum with fewer iterations.

QoS preserving learning. An unwanted pitfall of the learning-based approach in the existing literature is that the system needs to learn "bad" resource allocations that cause SLO violation by causing/creating these violations [12, 17–19]. While our approach too cannot completely eradicate the possibility of SLO violations, unlike prior works, we do not cause them intentionally. Instead, we adopt a QoS conservative approach where we start from with sufficient resource for all microservices to satisfy SLO, and then iteratively search for resource reduction opportunities based on the application's performance statistics. During the search/learning, PEMA always tries to maintain latency performance better than the SLO. Moreover, we dynamically tune how much resource we reduce based on how close our performance is to the SLO and stop tuning if the performance is at the SLO level. For example, with a response time SLO of 250ms, PEMA will try to reduce more resources when the response time is 150ms than when the response time is 200ms. Hence, during resource allocation navigation, PEMA does not set a resource allocation to violate the SLO intentionally.

Feedback-based navigation. Starting with ample resources for each microservices to comfortably satisfy SLO, PEMA uses the difference between current application performance and the SLO as an indicator of resource reduction opportunity. However, it does not tell us on which microservice(s) we should exercise the resource reduction. Hence, PEMA uses microservice-wise performance metrics to determine the target microservices. More specifically, PEMA uses the microservice-wise performance metrics to filter out the microservices approaching their bottleneck resource configuration and then implements a randomized selection process where the probability of picking a microservice is determined by its performance metrics. With unknown relation between a microservices resource allocation with the overall application performance, a guided randomized selection allows PEMA to explore various possible combinations of resource allocation.

3.2 Supporting Results for Design Rationales

Here, we provide corroborating observations for PEMA's design using our prototype microservices implementations. We first show why application's performance can be a safe yet effective indicator



(a) CDF of latency change (b) Response time change

Figure 7: (a) Distribution of end-to-end response time increase (normalized to SLO) due to monotonic resource reduction. (b) Change in response time (normalized to SLO) with resource (normalized to optimum).

of resource reduction, followed by how microservice-wise performance metrics can help PEMA navigate.

Gradual resource reduction for efficiency. In PEMA, we use the difference between SLO specified response time and current system response time to determine how much resource-saving opportunity is available. Our design choice is motivated by our observation that, in general, *monotonic resource changes across microservices result in monotonic changes in the end-to-end response time.* We say a resource reduction is monotonic if some microservice resources are decreased while other microservices' resources are unchanged. A resource change is not monotonic if some microservices receive greater resources while some others have their resource reduced, regardless of what happens to the aggregate resource allocation. Fig. 7(a) shows the CDF of increase in end-to-end response time for monotonic resource reduction for our applications. Note that there is no direct relationship between resource reduction and the amount of change in response time. This is because, the same amount of resource reduction on different microservices will have different impact on the end-to-end response time. The CDF is showing distribution of latency increase for random amounts of monotonic resource reduction on random numbers of microservices at random initial (before resource reduction) resource allocations. The CDF highlights the most likely impact of a monotonic resource reduction - an increase in the response latency regardless of the state of the microservice, i.e., its total resource allocation. The CDFs also show that the opposite, i.e., response latency decreasing with resource reduction, happens an only handful of times (10.2% for `TrainTicket` and 6.1% for `SockShop`). We attribute these cases as transient anomalies based on our observation of the application's performance metric fluctuations.

The key take away from Fig. 7(a) is that *by making monotonic resource reductions, we can gradually increase the latency to the SLO level.* In Fig. 7(b), we show examples of such monotonic resource reduction steps and its impact on latency. Here, we normalize the resource to the optimum resource allocation and the latency to the SLO level. PEMA's goal in Fig. 7(b) is to reach coordinate (1, 1) by gradually making monotonic resource changes. Note that the resource reduction steps in Fig. 7(b) is not unique. Moreover, monotonic resource reduction alone does not guarantee to reach the optimum resource allocation keeping the response latency within the SLO. Instead, it offers a QoS preserving approach of navigation to find efficient resource allocation.

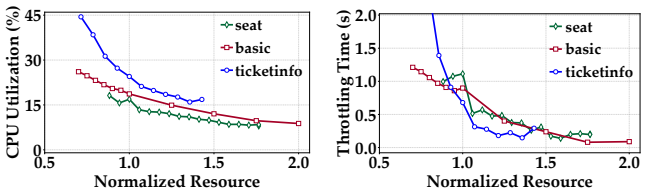
Table 1: Classification accuracy with CPU utilization and CPU throttling time as features to detect bottleneck microservices.

Microservice Name	Botteleneck Services	Accuracy (%)
TrainTicket	seat	94.18
TrainTicket	seat, ticketinfo	96.2
SockShop	carts	100.0
SockShop	carts, orders	98.3
HotelReservation	front-end	97.8
HotelReservation	front-end, search	95.6

Microservice-wise augmentation. While the response latency tells us about the resource reduction opportunities, it does not tell us from which microservices we should reduce the resources. We need to avoid microservices that may create a bottleneck during this resource reduction. We define a microservice's "bottleneck resource" as the resource allocation that makes the microservice a bottleneck. In PEMA, we use microservice-level performance metrics to identify the microservices with imminent bottleneck resources. However, as opposed to prior works where complex machine learning models are applied to determine such bottleneck services, we use only two performance metrics - CPU utilization and CPU throttling time [33].

Our choice of these performance metrics is based on our experiments. We intentionally create bottlenecks and use feature extraction to identify which performance metrics can be used to identify the bottleneck services reliably. Note that these experiments are done to assist in our design. PEMA does not need any offline experiments or pre-training. For each microservice, we collect the following performance metrics - `cpu_usage_seconds_total`, `memory_usage_bytes`, `cpu_cfs_throttled_seconds_total`, Jaeger tracing - `self_time`, and `duration`. We then run classification with various combinations of the performance metrics as features. We find that, when used as the classification features, CPU utilization and CPU throttling time give us the highest classification accuracy. Table 1 shows the classification accuracy for different applications with various bottleneck services.

To better understand the role of CPU utilization and CPU throttling time as bottleneck indicators, we track these metrics for three different microservices in `TrainTicket`- `seat`, `basic`, and `ticketinfo`, as we reduce their resources to create bottlenecks. To identify the bottleneck, we allocate sufficient resources to all other microservices. Fig. 8 shows the change in CPU utilization and CPU throttling as we reduce the resource of the microservice under investigation. We normalize the microservice resource allocations to their respective bottleneck resources. We make a few important observations here. *First,* the CPU utilization (Fig. 8(a)) changes gradually as the microservice approaches and eventually crosses the bottleneck resource. We also see that the utilization corresponding to bottleneck is different for different microservices. For example, ticketinfo's bottleneck utilization is around 25%, whereas seat's bottleneck utilization is around 15%. *Second,* CPU throttling time changes rapidly at bottleneck resource. The bottleneck CPU throttling time also varies with microservices.



(a) CPU utilization (b) CPU throttling time

Figure 8: Changes in CPU utilization and CPU throttling time with resource allocation for three bottleneck microsservices in `TrainTicket`- `seat`, `basic`, and `ticketinfo`.

3.3 PEMA

Here we present the details of PEMA's implementation that builds on our design principles and experimental observations.

Resource reduction opportunity. In PEMA, similar to gradient descent, we start with sufficient resources for all microservices and gradually decrease their resource based on how our resource change affects the end-to-end response time. We update resource allocation in regular intervals based on the response time observed in the previous interval. Since we rely on the response time statistics, we set sufficiently long update intervals to have stable response time statistics. For instance, in `TrainTicket`, `SockShop`, and `HotelReservation`, we use update interval of two minutes. For resource reduction at time step t, we first decide the number of microservices n^t to reduce resources from using

$$n^t = N \cdot \min\left(\frac{R - r^{t-1}}{\alpha R}, 1\right), \tag{3}$$

where $r^{t-1} = \mathcal{F}(\mathbf{x}^{t-1})$ is the response time in the previous time step. $\alpha \leq 1$ is a user-defined non-negative parameter that determines how aggressively we want to reduce the resource. A smaller α will reduce resource more aggressively and vice versa.

Next, using similar approach as Eqn.(3), we decide how much resource we reduce in the n^t microservices in percentage using

$$\Delta^t = \beta \cdot \min\left(\frac{R - r^{t-1}}{\alpha R}, 1\right) \cdot 100\%, \tag{4}$$

where $\beta \leq 1$ is another user defined parameter that decides the maximum resource reduction for any microservice in one time step. A high value of β makes PEMA aggressively change the resource between update intervals and vice versa. We analyze the impact of α and β in our evaluation in Section 4.3.

Using Eqns. (3) and (4), PEMA dynamically adjusts the amount of monotonic resource reduction as our response time r^t approaches SLO limit R. We can also set the values of α and β dynamically to have more aggressive reduction when $R - r^{t-1}$ is high and reduce the amount of reduction per interval as r^t approaches R. In addition, to avoid triggering resource change for transient perturbation in response time, we can keep a response time buffer by scaling down R, for instance, to 95%, in Eqns. (3) and (4).

Avoiding bottleneck services. For the i-th microservice, we denote its utilization as u_i with a bottleneck threshold U_i^{th} and CPU throttling time as h_i with a bottleneck threshold H_i^{th}. To

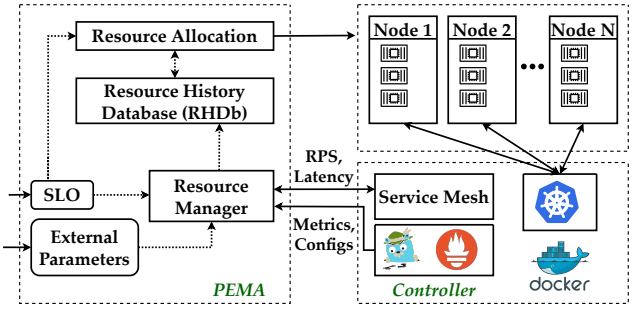


Figure 9: Block diagram of the PEMA.

decide the n^t candidate microservices, we first take the set of microservices that has a CPU throttling time less than their respective thresholds. We denote the set of indexes of these microservices as $\mathcal{I}^t = \{i : h_i^{t-1} \leq H_i^{th}\}$. We then normalize the utilization of each microservice in $\mathcal{I}^t$ to their respective utilization threshold as $u_i^{*t-1} = \frac{u_i^{t-1}}{U_i^{th}}$ and update the probability of each microservice in $\mathcal{I}^t$ as follows

$$p_i^t = 1 - \frac{u_i^{*t-1} - \min_{i \in \mathcal{I}^t}(u_i^{*t-1})}{1 - \min_{i \in \mathcal{I}^t}(u_i^{*t-1})} \tag{5}$$

Here, $\min_{i \in \mathcal{I}^t}(u_i^{*t-1})$ means the minimum normalized utilization among all the microservices in $\mathcal{I}^t$. Eqn.(5) indicates that a microservice with utilization equal to its threshold, i.e., $u_i^{*t-1} = 1$ will result in a "zero" probability ($p_i^t = 0$), whereas the microservice with the lowest utilization, i.e., $u_i^{*t-1} = \min_{i \in \mathcal{I}^t}(u_i^{*t-1})$, will have the probability of "one" ($p_i^t = 1$). We populate a new candidate set $\mathcal{I}^{*t}$ with a inclusion probability of p_i^t for the i-th microservice. If the size of $\mathcal{I}^{*t}$ is equal to or smaller than n^t, we take the entire set $\mathcal{I}^{*t}$ and reduce each microservice in $\mathcal{I}^{*t}$ and reduce their resource by Δ^t. However, if the size of $\mathcal{I}^{*t}$ is greater than n^t we uniformly randomly choose n^t microservices from $\mathcal{I}^{*t}$.

Dynamically updating bottleneck thresholds. As shown in Fig. 8, the bottleneck thresholds for utilization and CPU throttling time varies among microservices. Hence, we need to learn the appropriate threshold settings for each microservice. In PEMA, we begin with a conservative estimation of utilization threshold set at 15% and CPU throttling time threshold of "zero" (i.e., no CPU throttling) for all microservices. We expect all microservices to satisfy these thresholds as PEMA starts with ample resource allocation. Similar to our resource reduction approach, we opportunistically increase these thresholds. More specifically, at the beginning of every time step t, we update the utilization and CPU throttling time thresholds as follows

$$U_i^{th} = \max\left(U_i^{th}, u_i^{t-1}\right), \forall i \tag{6}$$

$$H_i^{th} = \max\left(H_i^{th}, h_i^{t-1}\right), \forall i \tag{7}$$

Iterative resource allocation. PEMA applies the resource reduction iteratively and saves all resource allocations, $\mathbf{x}^t$, and the response times, r^t, in a "resource allocation history database (RHDb)". The purpose of the RHDb is to allow PEMA to roll back to a previous SLO satisfying resource allocation for all microservices in case of an SLO violation. Even though the resource reduction slows down when the latency approaches the SLO, PEMA cannot guarantee that its opportunistic resource reduction will never cause an SLO violation. In addition, changes in microservice implementation or changes in its hardware configuration may also alter optimum resource allocation and cause SLO violations. In such cases, rolling back to a previous configuration allows PEMA to jump start on finding the new optimum, instead of resetting the resource allocation to the maximum and starting from scratch. While RHDb itself does not add significant overhead due to its lightweight single-table implementation, the action of rolling back may cause extra iterations for PEMA to find an efficient resource allocation. Nonetheless, the mechanism of roll back using RHDb is essential for PEMA's adaptability and QoS assurance.

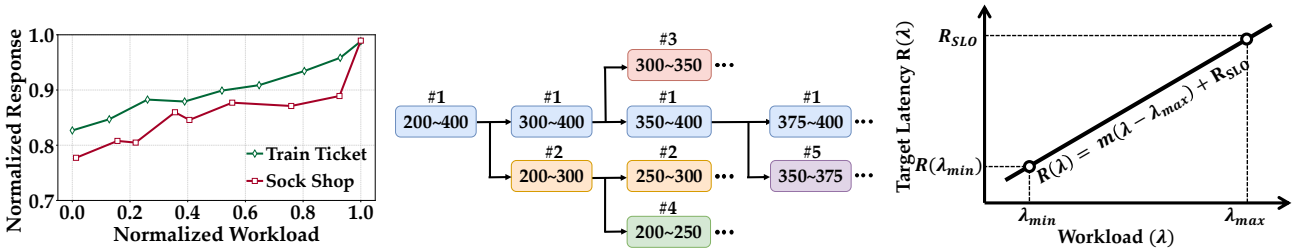
Escaping sub-optimum configurations. The combination of monotonic resource reduction and probabilistic choice of microservices to reduce resource may cause PEMA to make unfavorable resource reductions early on (e.g., making particular microservice reach bottleneck and push response time close to SLO) and settle at inefficient resource allocation, even though other microservices

Algorithm 1 PEMA

Input: SLO (R), affinity for resource reduction (α), maximum resource reduction limit (β), bottleneck utilization (U_i^{th}), and bottleneck CPU throttling time (H_i^{th}) for all microservices, exploration probability parameters A and B

Output: Resource allocation ($\mathbf{x}$)

1: **for** each time-step t **do**
2: **Performance metrics:** Collect end-to-end response time (r^{t-1}), CPU utilization u_i^{t-1}, and CPU throttling time h_i^{t-1}.
3: **Database update.** Insert x_i^{t-1}, r^{t-1}, U_i^{th}, and H_i^{th} to resource allocation history data base with key $t-1$.
4: **Handling SLO violation.** If $r^{t-1} > R$, update resource allocation to configuration from the resource allocation database with minimum resource and no SLO violation. Go to Line 11.
5: **Updating bottleneck thresholds.** For all microservices, update bottleneck thresholds for utilization, U_i^{th}, and CPU throttling time, H_i^{th}, following Eqns. (6) and (7), respectively.
6: **Exploration.** With a probability p_e^t defined in Eqn. (8), update resource allocation, $\mathbf{x}^t$ to a randomly chosen configuration from database without SLO violation. Go to Line 11.
7: **Resource reduction targets:** Determine number of microservice for resource reduction, n^t, using Eqn. (3) and resource reduction target for each microservice, Δ^t using Eqn. (4).
8: **Avoid bottleneck services:** Get the set $\mathcal{I}^t$ of microservices that do not exceed CPU throttling time threshold.
9: **Microservice-wise augmentation:** Build a new set $\mathcal{I}^{*t}$ from microservices in $\mathcal{I}^t$ with an inclusion probability of p_i^t defined in Eqn. (5).
10: **Resource reduction:** If $|\mathcal{I}^{*t}| > n^t$, uniformly randomly choose n^t microservices from $\mathcal{I}^{*t}$, else choose all microservices from $\mathcal{I}^{*t}$, and then update their resource to $x_i^{t-1} \cdot \Delta^t$.
11: **end for**



(a) Response time vs workload (b) Dynamic workload range (c) Dynamic response time target

Figure 10: (a) Response time change due to workload. (b) Dynamic workload range to bootstrap efficient resource allocation for different workloads. (c) Dynamically updating target response time to tackle response time change due to workload change.

have redundant resources. This can force PEMA to slow-down prematurely, even stop further resource reduction. To escape from such inefficient resource allocations, we implement random exploration where PEMA with a probability p_e^t rolls back to a uniformly random previous resource allocation in RHDb. We set p_e^t based on the response latency as follows

$$p_e^t = A \cdot \min\left(\frac{R - r^{t-1}}{\alpha R}, 1\right) + B \tag{8}$$

Here, A and B are exploration parameters that decide the maximum and the minimum probability of exploration, respectively, and satisfy $0 \leq B \leq A \leq 1$ and $A + B \leq 1$. The exploration probability decreases as PEMA's response time r^{t-1} approaches the SLO R. The random exploration also allows PEMA to "walk back" the resource reduction path it took and identify previously missed reduction opportunities. Naturally, the degree of exploration affects how quickly we reach an efficient resource allocation. Nonetheless, we do not anticipate this exploration to add significant overhead since PEMA can find an efficient resource allocation in a few tens of iterations.

Implementation of PEMA**.** We present the working principle in Algorithm 1 where PEMA takes performance metrics from the system using Prometheus and Linkered and then updates the resource allocation of the microservices while keeping a log of all resource allocations and response times in its database RHDb. The high-level architecture block diagram of PEMA is presented in Fig. 9.

3.4 Workload-Aware Resource Allocation

Our design of PEMA so far addresses how we can navigate to find an efficient resource allocation for our microservice-based application. Our design, through configuration rollback, can also handle changes in microservice implementation. Here we address how PEMA tackles the workload variations. For any cloud application, the workload intensity (i.e., requests per second) directly affects the response time, and hence, how much resource is needed [9–11]. In Fig. 10(a), we show the change in response time as the workload changes. As PEMA iteratively makes resource reductions based on the response time, a decrease in workload will falsely indicate resource reduction opportunities that do not work for high workloads, leading to many SLO violations when the workload increases. The same is true for prior ML-based approaches that do not explicitly address workload change [18, 19].

Hence, PEMA needs to identify efficient resource allocations at different workload levels. A straightforward way is to divide the workload variations into discrete workload ranges (e.g., a workload range from "X" requests-per-second to "Y" requests-per-second) and run multiple copies of PEMA in a "pseudo-parallel" fashion. We say pseudo-parallel as at any time only one PEMA is working on its corresponding workload range. Note here that the workload ranges need to be small enough to not significantly affect the response latency, requiring resource allocation changes, i.e., a single resource allocation should work for the entire range. For instance, a range of 25 requests-per-second in `TrainTicket` microservice is a suitable workload range.

Dynamic workload-range. While in principle multiple parallel PEMA works, it may take a long time to reach efficient allocations for every workload range. To accelerate the learning, we propose a novel approach where we start with a few (two/three) larger workload ranges and gradually split each range (i.e., parent range) into smaller ranges (i.e., child range) until we reach our target workload ranges. The goal here is to utilize learning from the parent ranges to bootstrap the learning process for the child ranges. During a range split, the parent range is divided into two equal child ranges. We attach PEMA of the parent range to the child range with a higher workload, whereas a new PEMA process is launched for the other child range. The new PEMA uses the resource allocations of the parent range as the starting point and requires fewer iterations to reach an efficient resource level. The intuition for this approach is that a resource allocation that satisfies SLO at a higher workload should also satisfy SLO for a lower workload. Fig. 10(b) illustrates the idea where we start with a workload range of 200~400 and then branch out to smaller ranges. The number on top of each range identifies the PEMA process attached to this range. The original PEMA process with id "#1" remains attached to the higher workload ranges (e.g., 300~400, 350~400, 375~400) as we split each range into smaller ranges.

Dynamic response time target. While this approach benefits the learning time, we need to tackle the latency variation due to workload changes when the workload ranges are large (e.g., 200~400 rps for `TrainTicket`). We use one PEMA process for each workload range, even during the initial stages with large ranges (e.g., PEMA #1 for 300~400 range in Fig. 10(b)). Each PEMA process needs to make an SLO preserving resource allocation that works for its entire range. To achieve this, instead of setting it to the SLO

specificity response time, we update R in Eqns. (3), (4), and (8) into a function of workload λ as follows

$$R(\lambda) = m \cdot (\lambda - \lambda_{max}) + R_{SLO} \tag{9}$$

Here, m is a parameter that determines the change in latency performance for a unit change in workload, λ_{max} is the upper limit of a workload range, and R_{SLO} is the SLO specified response time. Fig. 10(c) illustrates our approach of using a dynamic response time target. We see from Eqn. (9) that when the workload is low within a range, we set a conservative (i.e., lower than SLO) latency target to intentionally allocate more resource than needed and therefore allow headroom for higher workloads. This approach intentionally makes conservative inefficient resource allocations for lower workload levels within a range. However, as the ranges get smaller as we split them, the latency variation within a range also gets smaller, and so is the inefficiency. On the other hand, we learn m at the beginning of PEMA when we keep the resource allocation fixed for a few time steps while the workload changes. We then use linear regression on the workload vs response time (as in Fig. 10(a)) to extract m. Note that we learn m only once at the beginning when the workload ranges are large. During range splits, we keep the m from the parent range. Now, m may change as we make the resource allocations change on the microservice. Nonetheless, as our range split reaches the final workload ranges, we no longer need the dynamic response target, and m becomes irrelevant.

3.5 Handling Transient Events

From our extended experiments we identify that PEMA is susceptible to unnecessary SLO violations due to transient dips in the response time. More specifically, after PEMA has already identified an efficient allocation, a momentary/transient dip in response time drives PEMA to make resource reductions only to meet with SLO violation in the next iteration. To circumvent this, we adopt a moving average approach where we take the average of the response time of K recent time steps and update the n^t and Δ^t as follows

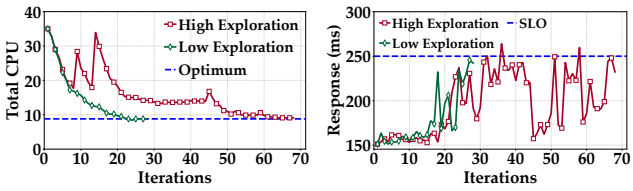
$$n^t = N \cdot \min\left(\frac{R - \frac{1}{K}\sum_{k=1}^{K} r^{t-k}}{\alpha R}, 1\right) \tag{10}$$

$$\Delta^t = \beta \cdot \min\left(\frac{R - \frac{1}{K}\sum_{k=1}^{K} r^{t-k}}{\alpha R}, 1\right) \cdot 100\% \tag{11}$$

Note that, to ensure QoS, we do not apply this moving averaging for detecting SLO violations. We still roll back resource allocations based on the most recent response time as in Line 4 in Algorithm 1.

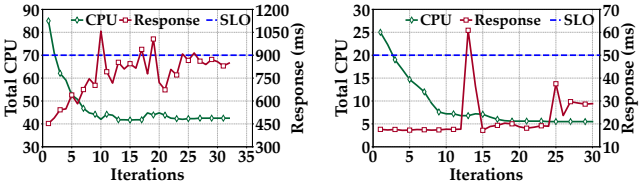
4 EVALUATION

We use our microservice application prototypes, `TrainTicket`, `SockShop`, and `HotelReservation`, to evaluate PEMA. Here we first discuss details of PEMA's execution followed by performance evaluation against other resource allocation strategies. We then present how different parameters affect PEMA, and finally show how PEMA can adapt to change in operating conditions.



(a) CPU allocation (b) Response time

Figure 11: Execution of PEMA on SockShop with different explorations. The exploration parameters in Eqn. (8) for high exploration are $A = 0.1, B = 0.01$, and for low exploration are $A = 0.05, B = 0.005$.



(a) `TrainTicket` (b) `HotelReservation`

Figure 12: Execution of PEMA for `TrainTicket` and `HotelReservation`.

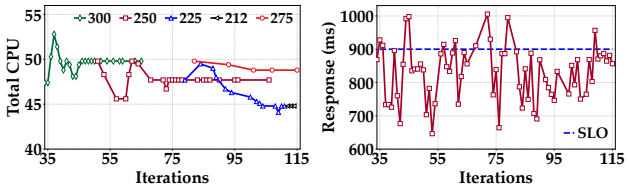
4.1 Execution of PEMA

Here, we first show how PEMA finds efficient resource allocation using iterative resource reduction, where the duration of each iteration is two minutes. We then demonstrate how workload-aware PEMA utilizes the dynamic workload range and response time target. Finally, we present a 36-hour long experiment with PEMA making efficient resource allocation maintaining QoS.

Efficient resource allocation. Fig. 11(a) demonstrates the iterative resource allocation and Fig. 11(b) shows the corresponding response times for `SockShop` under a workload of 700 requests per second for two different sets of exploration parameters. Here, the optimum total CPU allocation is 8.8 which is identified using extensive trial and error.

We see in Fig. 11(a) that when a higher exploration is used, PEMA intentionally increases the resource allocation twice around iteration 10 by going back to an older and higher CPU allocation. We also see that PEMA with high exploration settles at an inefficient allocation after 20 iterations as the response time reaches SLO (Fig. 11(b)). However, due to exploitation, we see that around iteration 45, it rolls back to an older allocation and finds its way to the efficient allocation. Incidentally, PEMA with low exploration also reaches the efficient resource allocation. We see a few SLO violations in Fig. 11(b) which are mitigated immediately by increasing the CPU resource. Figs. 12(a) and 12(b) show the iterative resource change and the corresponding response times for `TrainTicket` and `HotelReservation`, respectively.

Regardless of the microservice implementation, we see that PEMA can successfully find efficient resource allocations with only a few unintentional SLO violations.



(a) Total CPU allocation over different ranges (b) Response time of the iterations

Figure 13: Execution of PEMA on `TrainTicket` with dynamic workload range. (a) CPU allocation. (b) Response time.

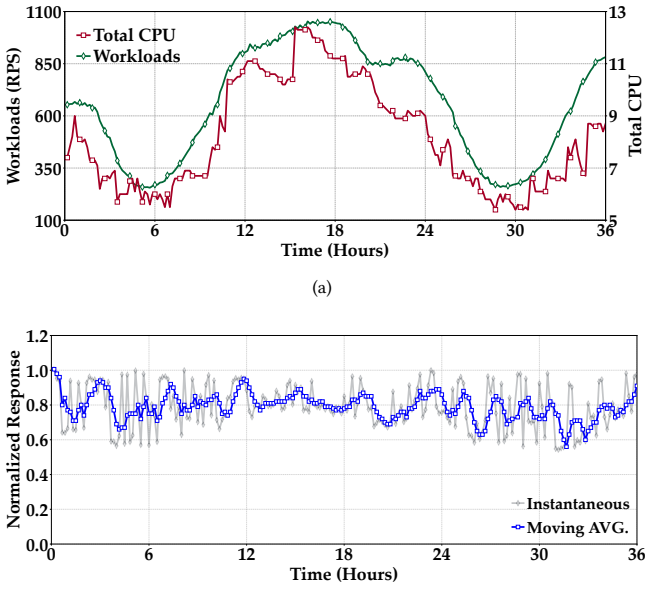


Figure 14: Extended execution of PEMA in SockShop. (a) Workload and CPU allocation. (b) Response time normalized to SLO.

Dynamic workload range. Next, in Fig. 13(a), we show the resource allocation of PEMA for `TrainTicket` as our workload varies between 200 and 300 requests per second. The legend in this figure indicates the upper limit on the workload range. The workload range 300 (i.e., 200~300) first splits into ranges 300 and 250 around iteration 50. The 250 range splits into 250 and 225 around iteration 80, while the 300 splits into 300 and 275 right before iteration 85. We see that each workload range finds an efficient allocation within a few iterations as they start from an already good allocation. Fig. 13(b) shows the corresponding the response time. We see some SLO violations, which are mitigated by PEMA.

Extended execution. We run PEMA on SockShop for 36-hour where we change the workloads between 200 and 1100 requests per second following the workload pattern of Wikipedia collected from [34]. Fig. 14(a) shows the workload pattern and the corresponding resource allocation. We see that PEMA varies the total resource

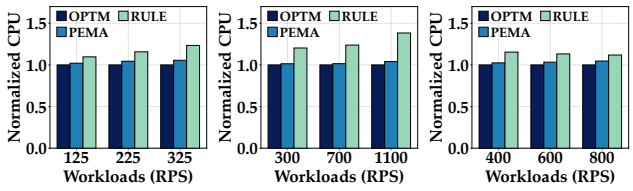


Figure 15: Performance comparison of PEMA against optimum (OPTM) and commercial autoscaler (RULE). The CPU allocation is normalized to that of OPTM. PEMA is close to optimum and saves up to 33% resource compared to RULE.

allocation with changing workload to maintain efficient allocation. Note here that simply varying scaling resource allocation based on workload does not work on microservices as the distribution of the resource plays an important role in performance. Fig. 14(b) shows the corresponding response times. We show both the instantaneous (i.e., most recent) and moving average responses with a window size of five. Recall that PEMA reduces resources based on the moving average to avoid transient changes while tackling SLO violation based on the instantaneous response time.

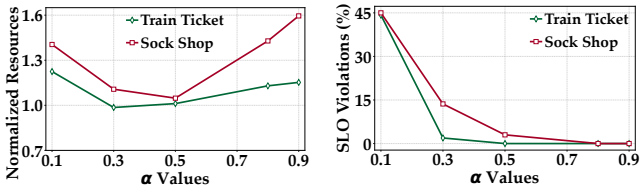
4.2 Performance evaluation

Benchmark strategies. We compare the resource allocation efficiency of PEMA against two benchmark strategies - optimum (OPTM) and rule-based (RULE). In OPTM, we use an exhaustive trial and error search to identify the best possible resource allocation. We identify a resource allocation as optimum if a small resource reduction (in our case 0.1 CPU) in any of the microservices results in a SLO violation. Note that, OPTM cannot be used in practice as it causes many SLO violations during trial and error. It acts as the upper limit of resource efficiency achievable by any resource manager. RULE is Kubernetes' rule-based resource scaling [35]. We chose RULE as a commercially available resource allocation algorithm to gauge PEMA's efficiency improvement. We do not compare PEMA to the ML-based resource allocation strategies as they do not focus on resource allocation efficiency.

Comparison of resource allocation efficiency. We run each of the three microservices applications using PEMA and the two benchmark algorithms. Since OPTM requires extensive manual search, we evaluate these algorithms for three different workload levels for each microservice. Also, since PEMA is provably efficient, we run PEMA several times under each setting and show the average resource allocation. We normalize each resource allocation for each workload level using the resource allocation of OPTM.

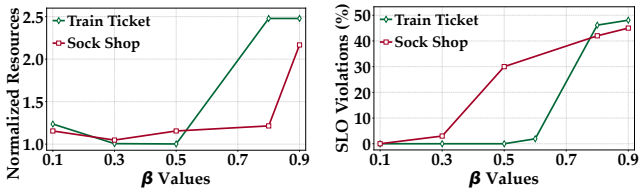
Figs. 15(a), 15(b), and 15(c) show the resource allocations of `TrainTicket`, SockShop, and `HotelReservation`, respectively for the three different algorithms. We see that PEMA's resource allocation efficiency is very close to OPTM. We also observe that PEMA's efficiency drifts away with increasing workload. On the other hand, PEMA consistently beats RULE, saving as much as 33% on resource allocation for SockShop at high workloads.

The performance comparison results demonstrate that despite being a lightweight resource manager, PEMA can deliver close to optimum resource allocation while retaining its capability to



(a) Resource allocation (b) SLO violations

Figure 16: PEMA's sensitivity to α for a $\beta = 0.3$ (a) Resource allocation normalized to optimum. (b) SLO violations.



(a) Resource allocation (b) SLO violations

Figure 17: PEMA's sensitivity to β for a $\alpha = 0.5$ (a) Resource allocation normalized to optimum. (b) SLO violations.

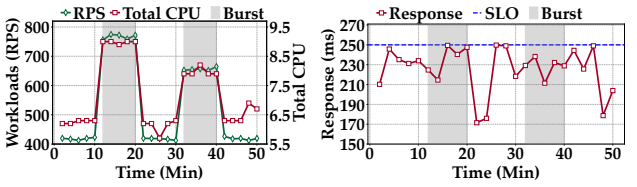
tackle workload variation without any significant overhead (e.g., ML training).

4.3 Parameter Sensitivity

Here we study how the two parameters α and β affect PEMA. Recall that α in Eqn. 3 determines how aggressively we reduce resource - smaller α makes PEMA reduce more resource for the same difference between response time and SLO. β, on the other hand, determines the maximum percentage resource reduction in each resource update iteration - smaller β results in smaller resource change and vice versa. For this study, we run experiments on `TrainTicket` and `SockShop` with workload 225 and 700 requests per second.

In Fig. 16(a), we show the change in resource allocation and in Fig. 16(b), we show the number of SLO violations as we change α. During this experiment, we keep $\beta = 0.3$. We see that both smaller and larger values of α result in sub-optimal resource allocations for `TrainTicket` and `SockShop`. This is because, for small α, PEMA is too aggressive making many SLO violations (as seen in Fig. 16(b)) and force to revert back to inefficient allocations. For high α, on the other hand, PEMA is slowed down prematurely at inefficient allocations, although it suffers much fewer SLO violations.

Next, in Figs. 17(a) and 17(b), we show the impact of change in β while we keep $\alpha = 0.5$. Similar to our observation for α we see that aggressive resource reduction due to higher values of β results in sub-optimal resource allocation while also suffering from many SLO violations. While PEMA is somewhat sensitive to both α and β, we can set α and β for any system by tuning based on SLO violation. We can take a conservative approach, start with large α and small β, and gradually change their values keeping a close eye on the SLO violations.



(a) (b)

Figure 18: Operation of PEMA with bursty workload in SockShop. (a) Workload and CPU allocation. (b) Response time.

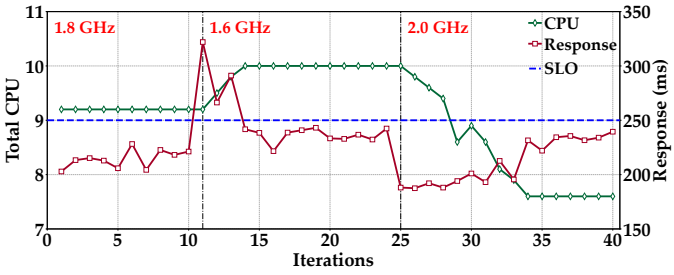


Figure 19: Adaptability of PEMA to changes in CPU speed for SockShop. The CPU speed change represents hardware or software updates that alters the resource demand.

4.4 Adaptability

Workload bursts. PEMA can seamlessly handle sudden changes in workload. In Fig. 18, we show how PEMA handles workload bursts for SockShop by switching the resource allocation to the workload range corresponding to the workload burst. Here, we consider PEMA has already traversed through the resource reduction iterations for all workload ranges. As shown in Fig. 18(a), we create two workload burst of 10 minutes where the workload shoots up from 400 RPS to around 750 RPS and 650 RPS. We see that PEMA quickly changes the CPU allocation to keep the response time below SLO (in Fig. 18(b)). Note here that, since we update the resource allocation every two minutes, PEMA can react to a workload burst lasting less than two minutes. Nevertheless, we can adapt PEMA to respond to short-lived workload bursts by reducing the resource update interval.

Operating environment. Our PEMA's lightweight design enables adaptability to operation condition changes. Such changes may lead to different response times even when the resource allocation is not altered. We change our server's CPU clock speeds from 1.8 GHz to 1.6 GHz and 2 GHz. These changes mimic a real-world scenario where a hardware or software change in the microservice alters the resource allocation dynamics. While we make the clock speed changes, we use PEMA to manage SockShop's resource. A change in CPU frequency essentially changes the resource requirement for satisfying the SLO. Fig. 19 shows the CPU allocation and the corresponding response time as we change the CPU frequency. We see that PEMA can successfully change the resource allocation to satisfy the SLO demonstrating its capabilities to adapt.

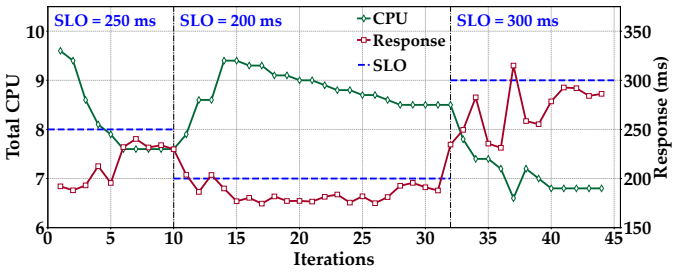


Figure 20: Adaptability of PEMA to changes in SLO for SockShop. Dynamic SLO can be used to trade performance for resource savings.

Dynamic SLO change. In Fig. 20, we show that PEMA can also navigate towards efficient resource allocation as we change the SLO. Dynamically changing SLO can be a useful approach for applications that are willing to trade performance for resource savings to meet long-term goals such as cost budget [36]. Dynamic SLO essentially adds another control knob for managing the microservices application. Unlike existing ML-based microservice managements, which will need to retrain with new SLO, PEMA can quickly adapt to SLO changes and tune the resource accordingly.

5 RELATED WORKS

Microservice autoscaling. Resource autoscaling has been extensively studied in the public cloud domain [9, 37–40]. The recent advancement of microservices has attracted a similar interest in autoscaling of microservice-based applications in academic settings [41, 42], as well as industrial settings [10, 20]. These autoscalers implement rule-based approaches in resource management. For example, Kubernetes [20] uses 90-th percentile resource usage in recent samples to set CPU and memory allocations with a 15% overprovisioning. Google Autopilot [10] uses 95-th percentile for CPU and maximum for memory in the recent samples as a marker for resource allocation in the upcoming interval. Alternative to the rule-based approach, Google also uses ML-based autoscaling using a combination of reinforcement learning and time series analysis [43]. [44] also proposes rule-based autoscaling based on CPU and memory utilization. However, rule-based autoscaling requires deep application knowledge to set up the thresholds that can vary with application. Meanwhile, [41] proposes hybrid autoscaling based on analytical modeling using a layered queue network.

SHOWAR [45], in spirit, is the closest to our design approach. It uses the variance in historical usage for vertical scaling and a proportional-integral-derivative (PID) controller for horizontal scaling. Nonetheless, SHOWAR still requires extensive tracing from the CPU scheduler for its scaling decision. On the other hand, similar to our opportunistic resource reduction, [46] utilizes "resource deflation" where preemptible virtual machines' resources are dynamically controlled. However, while resource deflation gives away transient resources to avoid preemption, we use resource reduction as a mean to find efficient allocation by carving redundant resources.

SLO oriented resource management. In another line of work, ML-based approaches are used to identify and mitigate root causes of SLO violations in microservices [4, 12, 17–19]. For example, Sinan [17] uses a neural network to estimate short-term performance and a boosted trees model to estimate long-term performance to make per tier resource allocation. Sinan allows SLO violations to identify corner cases for resource allocation. Seer [19] requires fine-grained tracing for building its model and SLO violating cases to train its deep neural network to identify QoS violations. AlphaR [4], on the other hand, uses neural graph networks to capture the complex relationship between microservices and estimate application performance for resource allocation. Despite their impressive results in capturing minute details of microservices, they heavily depend on data and are slow to dynamically changing conditions for microservices. In designing PEMA, we depart from using complicated ML models and instead trade capturing microservice details for agility and adaptability in resource management.

6 CONCLUDING REMARKS

In this paper, we proposed PEMA, an iterative feedback-based approach to autoscaling microservices. PEMA is lightweight as it only requires the applications end-to-end performance and microservice-level CPU utilization and CPU throttling to navigate to efficient microservice resource allocation. Utilizing the lightweight design, we also developed a novel approach of dynamic workload-ranging to make workload-aware resource allocation with PEMA. Using three prototype microservice implementations, we showed that PEMA can achieve a performance close to the optimum resource allocation and save as much as 33% resource compared to commercially used rule-based resource allocation.

Limitations of PEMA's current implementation. PEMA's implementation has several limitations that we plan to address in its future iterations. First, when PEMA causes an unintentional SLO violation, it rolls back the resource configuration in the next time step. Hence, the application suffers from bad performance during the entire resource update interval (e.g., 10 minutes). PEMA can be improved by implementing higher resolution performance monitoring (e.g., within 10 seconds), catching the SLO violations early, and rolling back configuration to mitigate it. Further, PEMA rolls back the configuration to the most recent configuration without SLO violation. It does not take into account the degree of SLO violation. For instance, a QoS violation where the response time is significantly higher than the SLO indicates that PEMA should roll back the configuration farther into the past to allocate more resources. On the other hand, while PEMA logs the resource allocation of all microservices and response times in its allocation history database, RHDb, for rollback and exploration purposes, it does not utilize this information in its decision. Finally, PEMA in this study only considers CPU resource allocation meanwhile memory and I/O resources allocation can also be important for microservices' performance depending on the nature of the application. Moreover, PEMA also does not explicitly address the impacts and trade-offs among vertical (i.e increasing resource in one node) and horizontal (i.e., increasing the number of nodes) resource scaling.

7 ACKNOWLEDGMENTS

This work is supported in parts by the US National Science Foundation under grant number CNS-2104925.

REFERENCES

[1] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 155–158, 2018.

[2] Y. G. et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *ASPLOS*, 2019.

[3] R. e. a. Heinrich, "Performance engineering for microservices: research challenges and directions," in *ICPE*, pp. 223–226, 2017.

[4] X. H. et al., "Alphar: Learning-powered resource management for irregular, dynamic microservice graph," in *IPDPS*, 2021.

[5] E. Wolff, *Microservices: flexible software architecture.* Addison-Wesley Professional, 2016.

[6] "The definition of microservice." https://martinfowler.com/microservices/, 2022. Accessed: 01/26/2022.

[7] "Introduction to microservices." https://www.nginx.com/blog/introduction-to-microservices. Accessed: 01/20/2022.

[8] H. Z. et al., "Overload control for scaling wechat microservices," in *SoCC*, 2018.

[9] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems*, vol. 30, 2012.

[10] "Google cloud autoscale." https://cloud.google.com/compute/docs/load-balancing-and-autoscaling. Last Accessed: 01/05/2022.

[11] "Azure autoscale." https://azure.microsoft.com/en-us/features/autoscale/. Last Accessed: 01/05/2022.

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

[13] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[14] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[15] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *ISCA*, 2014.

[16] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ISCA*, 2015.

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

[18] Y. G. et al., "Sage: practical and scalable ml-driven performance debugging in microservices," in *ASPLOS*, 2021.

[19] Y. G. et al., "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *ASPLOS*, 2019.

[20] "Kubernetes autoscaler." https://github.com/kubernetes/autoscaler. Last Accessed: 01/27/2022.

[21] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[22] "Sock shop microservice demo." https://microservices-demo.github.io/. Accessed: 08/31/2021.

[23] "Docker: Empowering app development for developers." https://www.docker.com/. Accessed: 01/20/2022.

[24] "Kubernetes: Production grade container orchestration." https://kubernetes.io/. Accessed: 01/20/2022.

[25] "Practical efficient microservice autoscaling." https://github.com/rajibhossen/microservice-autoscaling.

[26] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," pp. 323–324, 2018.

[27] "grpc: A high performance, open source universal rpc framework." https://grpc.io/. Accessed: 01/20/2022.

[28] "Prometheus - from metrics to insights." https://prometheus.io/. Last Accessed: 10/08/2021.

[29] "Linkerd: A different kind of service mesh." https://linkerd.io/. Accessed: 01/15/2022.

[30] "Jaeger - end to end tracing distributed tracing." https://www.jaegertracing.io/. Last Accessed: 10/08/2021.

[31] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *ICPE*, 2019.

[32] "Amazon aws autoscale." https://docs.aws.amazon.com/autoscaling/index.html. Last Accessed: 01/05/2022.

[33] "Kubernetes cpu throttling." https://vmblog.com/archive/2021/10/07/kubernetes-cpu-throttling-the-silent-killer-of-response-time-and-what-to-do-about-it.aspx. Accessed: 10/26/2021.

[34] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[35] "Kubernetes horizontal pod autoscaler." https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Last Accessed: 01/05/2022.

[36] M. A. Islam, S. Ren, A. H. Mahmud, and G. Quan, "Online energy budgeting for cost minimization in virtualized data center," *IEEE Transactions on Services Computing*, vol. 9, no. 3, pp. 421–432, 2015.

[37] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," in *ACM Comput. Surv.*, vol. 51, (New York, NY, USA), Association for Computing Machinery, July 2018.

[38] A. F. Baarzi, T. Zhu, and B. Urgaonkar, "Burscale: Using burstable instances for cost-effective autoscaling in the public cloud," in *SoCC*, 2019.

[39] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *OSDI*, 2018.

[40] M. Wajahat, A. A. Karve, A. Kochut, and A. Gandhi, "Mlscale: A machine learning based application-agnostic autoscaler," *Sustain. Comput. Informatics Syst.*, vol. 22, pp. 287–299, 2019.

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

[42] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *ACSOS*, 2020.

[43] K. R. et al., "Autopilot: Workload autoscaling at google scale," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[44] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *ICDCS*, 2019.

[45] A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," in *SoCC*, 2021.

[46] P. Sharma, A. Ali-Eldin, and P. Shenoy, "Resource deflation: A new approach for transient resource reclamation," in *EuroSys*, 2019.