Practical Efficient Microservice Autoscaling

Md Rajib Hossen
University of Texas at Arlington

Mohammad A. Islam
University of Texas at Arlington

1. INTRODUCTION

Motivation. In recent years, the adoption of microservices in production systems has been steadily growing. With their loosely-coupled and lightweight components, microservices are easier to manage than traditional monolithic applications. However, microservices introduce new challenges towards efficient resource management because their high number of individually manageable components creates a large configuration space. Moreover, microservices have complex inter-dependencies, making identifying "good" configurations harder. Even the same amount of resources can result in different response latencies based on how the resource is distributed among different microservices. Meanwhile, existing cloud management approaches are designed for monolithic deployments and cannot capture the complex interactions between microservices.

Limitation of state-of-art approaches. Recently, several Machine Learning (ML) based solutions have been proposed where an ensemble of ML techniques are used to extract the complex relationship between microservices resources and performance [1, 2, 3, 4, 5]. However, these ML-based approaches are fundamentally limited by their dependency on extensive training on high-resolution data (e.g., request level traces) to capture microservice dynamics. More importantly, to identify and mitigate the root causes of Service Level Objective (SLO) violations, ML-based solutions intentionally cause or allow SLO violations that are undesirable in commercial applications [1, 2, 4, 3]. Further, time-consuming ML retraining may be triggered frequently due to changes in microservices dynamics caused by software updates and migration to servers with different hardware.

Our contributions. To circumvent the aforementioned limitations, we develop PEMA (**P**ractical **E**fficient **M**icroservice **A**utoscaling), a lightweight resource manager which does not rely on extensive training. Instead, PEMA iteratively interacts with the microservice application to find efficient resource allocations. In this work, we mainly consider CPU allocation, which significantly impacts microservices due to their stateless nature [2]. PEMA*'s goal is not to directly provide efficient resource configuration based on the system's state, rather it is designed to navigate the configuration space to quickly find an efficient resource allocation.*

To avoid intentional SLO violations, PEMA starts with sufficient resource for all microservices and then focuses on finding *resource reduction opportunities* while ensuring the end-to-end performance i.e., SLO is not violated. Since we start from ample resource, even when PEMA performs poorly (i.e., fails to identify the resource reduction opportunities), the microservices retains enough resource, albeit inefficient, to satisfy the SLO. To facilitate the opportunistic resource reduction-based approach, we adopt a "monotonic resource change" strategy where we do not mix resource reduction (for some microservices) and resource increase (for some other microservices) in the same iteration of resource change. We observe that, despite the complex interdependencies between microservices, monotonic resource change results in a monotonic change in applications response time. This allows us to use the response time as an indicator of resource reduction opportunity and make gradual resource reduction to navigate towards an efficient resource allocation. In addition, experiments on our prototype microservice implementation reveal that using only two microservice-level performance metrics - CPU utilization and CPU throttling, PEMA can avoid resource reduction from microservices approaching their bottleneck resource.

Our performance evaluation on three prototype microservice implementations demonstrate that PEMA can attain resource efficiency close to the optimum with a few tens of iterations. We show that PEMA can save as much as 33% resource compared to rule-based resource allocation strategies of commercially available cluster managers such as the Kubernetes.

2. DESIGN OF PEMA

Problem formulation. 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, a practical microservices resource manager that finds a provably efficient solution to ORA.

Solution approach of PEMA. Solving PEMA can be interpreted as tuning the application resources that will make the response latency exactly equal to the SLO specified level. Starting with ample resources for each microservices, PEMA uses the difference between current application performance and the SLO as an indicator of resource reduction opportunity. To identify the target microservices for resource reduction, PEMA uses microservice-wise performance metrics. More specifically, PEMA uses the CPU utilization and CPU throttling 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 CPU utilization.

Resource reduction. 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. Next, 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. The parameters α and β determines how aggressively we reduce the resource in every time step.

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 as h_i with a bottleneck threshold H_i^{th}. To decide the n^t candidate microservices, we first take the set of microservices that has a CPU throttling 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 $\mathcal{I}^t$. 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 $\mathcal{I}^{*t}$.

Updating bottleneck thresholds. We use online learning to find the bottleneck thresholds for utilization and CPU throttling for each microservice. In PEMA, we begin with a conservative estimation of utilization threshold set at 15% and CPU throttling threshold of "zero" (i.e., no throttling) for all microservices. Now as we gradually reduce the resource, the utilization and throttling of all microservices eventually crosses their respective threshold limits. However, after any time step $t-1$, if PEMA sees that there was no SLO violation, it updates all the threshold limits for time step t 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 database is to allow PEMA to move back to a previous resource allocation in case of an SLO violation and enable random exploration.

Algorithm 1 PEMA

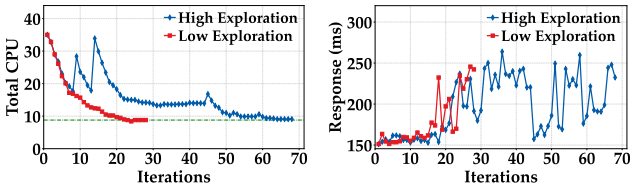
Input: SLO (R), affinity for resource reduction (α), maximum resource reduction limit (β), bottleneck utilization (U_i^{th}), and bottleneck CPU throttling (H_i^{th}) for all microservices, exploration probability parameters A & B
Output: Resource allocation ($\mathbf{x}$)

1: **for** each time-step t **do**
2: **Performance metrics:** Collect end-to-end response time (r^{t-1}), microservice utilization u_i^{t-1}, CPU throttling 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, 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 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**

Escaping sub-optimum configurations. PEMA may make unfavorable resource reductions at the beginning (e.g., making particular microservice reach bottleneck) and settle at inefficient resource allocation, even though other microservices have ample resources. 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

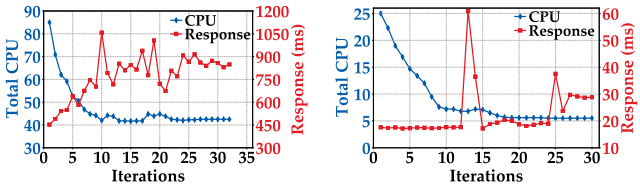
$$p_e^t = A \cdot \min\left(\frac{R - r^{t-1}}{\alpha R}, 1\right) + B \tag{8}$$

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. Algorithm 1 summarizes the iterative steps and algorithm flow of PEMA.



(a) CPU allocation (b) Response time

Figure 1: Execution of PEMA on Sock Shop with different explorations. The exploration parameters in Eqn. (8) for high exploration are $A = 0.1, B = 0.01$, and for low exploration, $A = 0.05, B = 0.005$.



(a) Train Ticket (b) Hotel Reservation

Figure 2: PEMA on Train Ticket and Hotel Reservation.

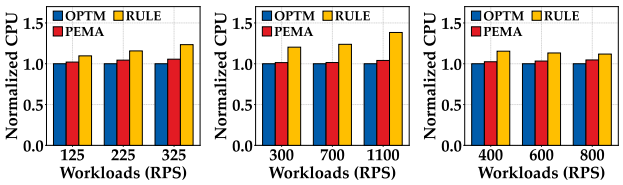
3. EVALUATION

Experimental methodology. We use three widely-used prototype microservices for our evaluation [1, 2, 3]. We implement Train Ticket consisting of 41 microservices, Sock Shop with 13 microservices, and Hotel Reservation with 18 microservices. We deploy these services in Docker containers managed by Kubernetes. 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 Ubuntu 20.04.3 operating system. We set $\alpha = 0.5$ and $\beta = 0.3$. The SLOs (R) for Train Ticket is $900ms$, for Sock Shop is $250ms$, and for Hotel Reservation is $50ms$.

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. It acts as the upper limit of resource efficiency achievable by any resource manager. RULE is Kubernetes' rule-based resource scaling. 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.

Execution of PEMA. Fig. 1(a) demonstrates the iterative resource allocation and Fig. 1(b) shows the corresponding response times for Sock Shop under a workload of 700 requests per second for two different sets of exploration parameters. We see a few SLO violations in Fig. 1(b) which are mitigated immediately by increases in CPU resource. Figs. 2(a) and 2(b) show the iterative resource changes and the corresponding response times for the microservices Train Ticket and Hotel Reservation, respectively.

Comparison of resource allocation efficiency. We run each of the three microservices applications using PEMA and the two benchmark algorithms. We normalize each resource allocation for each workload level using the resource allocation of OPTM. Figs. 3(a), 3(b), and 3(c) show the resource allocations of Train Ticket, Sock Shop, and Hotel Reservation, 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 from OPTM with increasing workload. On the other hand, PEMA consistently beats saving as much as 33% on resource allocation for Sock Shop at high workloads.



(a) Train Ticket (b) Sock Shop (c) Hotel Reserv

Figure 3: Performance comparison of PEMA against optimum (OPTM) and commercial autoscaler (RULE).

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

4. CONCLUDING REMARKS

In this paper, we proposed PEMA, an iterative feedback-based approach to autoscale microservices. PEMA is lightweight as it only requires the application's end-to-end performance and microservice-level CPU utilization and CPU throttling to navigate to efficient microservice resource allocation.

Limitations of PEMA's approach. 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. Also, due to a randomized exploration based search, PEMA offers provably efficient management and can result in arbitrarily inefficient resource allocations at times.

Future work. PEMA's implementation has several limitations that we plan to address. First, after an unintentional SLO violation, PEMA rolls back to a previous configuration in the next time step (Line 4 in Algorithm 1). Hence, the application suffers from bad performance during one full update interval. Second, PEMA does not take into account the extent of SLO violation during roll back, rather it rolls back to the most recent SLO satisfying configuration. Third, we do not actively utilize the RHDb which logs resource allocation and performance data that can offer insights into the behavior of the microservice application. Finally, PEMA in this study only considers CPU resource allocation and does not explicitly address vertical and horizontal scaling.

5. REFERENCES

[1] 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.

[2] 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.

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

[4] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *ASPLOS*, 2019.

[5] X. Hou, C. Li, J. Liu, L. Zhang, S. Ren, J. Leng, Q. Chen, and M. Guo, "Alphar: learning-powered resource management for irregular, dynamic microservice graph," in *IPDPS*, 2021.