SHOWAR: Right-Sizing And Efficient Scheduling of Microservices

Ataollah Fatahi Baarzi
The Pennsylvania State University
ata@psu.edu

George Kesidis
The Pennsylvania State University
gik2@psu.edu

Abstract

Microservices architecture have been widely adopted in designing distributed cloud applications where the application is decoupled into multiple small components (i.e. "microservices"). One of the challenges in deploying microservices is finding the optimal amount of resources (i.e. size) and the number of instances (i.e. replicas) for each microservice in order to maintain a good performance as well as prevent resource wastage and under-utilization which is not cost-effective. This paper presents *SHOWAR*, a framework that configures the resources by determining the number of replicas (horizontal scaling) and the amount of CPU and Memory for each microservice (vertical scaling). For vertical scaling, *SHOWAR* uses empirical variance in the historical resource usage to find the optimal size and mitigate resource wastage. For horizontal scaling, *SHOWAR* uses basic ideas from control theory along with kernel level performance metrics. Additionally, once the size for each microservice is found, *SHOWAR* bridges the gap between optimal resource allocation and scheduling by generating affinity rules (i.e. hints) for the scheduler to further improve the performance. Our experiments, using a variety of microservice applications and real-world workloads, show that, compared to the state-of-the-art autoscaling and scheduling systems, *SHOWAR* on average improves the resource allocation by up to 22% while improving the 99th percentile end-to-end user request latency by 20%.

CCS Concepts: • **Computer systems organization → Cloud computing**.

Keywords: autoscaling, microservices, cloud computing

ACM Reference Format:
Ataollah Fatahi Baarzi and George Kesidis. 2021. *SHOWAR*: Right-Sizing And Efficient Scheduling of Microservices. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472883.3486999

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486999

1 Introduction

The microservices architecture is a recent and increasingly popular paradigm for designing interactive and user-facing services where hundreds of small and fine-grained components (i.e. "microservices") collectively work on serving end-user requests in a distributed setting [1, 4, 10, 16, 35]. Breaking down an application into small microservices brings several benefits. It allows different developer teams to independently work on (possibly technologically) different microservices [61]. Also, each microservice can scale and operate independently depending on its own state and incoming workload which results in better performance and reliability of the application as a whole [63]. Finally, a microservices architecture may facilitate debugging for performance and correctness issues [54].

One of the important challenges in deploying microservices is cluster resource allocation and scaling. To meet some performance and reliability goals, before deploying the microservices, the developers (application owners) have to specify the size of each microservice in terms of compute resources such as CPU and memory (i.e. "vertical sizing"), as well as the number of instances (or "replicas") for each microservice (i.e. "horizontal sizing"). However, the resource needs of microservices will depend on potentially complex demand processes, and may be difficult to predict a priori. On the one hand, allocating more compute resources than what is required for the microservices to operate normally leads to low cluster resource utilization which is not cost effective [51, 56]. On the other hand, allocating less resources than what is needed for the microservices can lead to performance degradation and service unavailability both of which can result in revenue loss for the application owner [11].

In this paper, we present *SHOWAR*, a system designed for both horizontal and vertical autoscaling of microservices managed by Kubernetes [3], the state of the art container orchestrator platform. For vertical autoscaling, *SHOWAR* embraces the variance in the historical resource usage to find the optimal resource sizing of each microservice to maintain a good performance while avoiding low resource utilization. For horizontal autoscaling, *SHOWAR* uses metrics from Linux kernel thread scheduler queues (in particular eBPF *runq latency*) as its autoscaling signal to make more accurate and meaningful autoscaling decisions. At its core, *SHOWAR* uses

basic ideas from control theory to control the number of replicas for each microservice based on signals from the microservice's run-time. In particular, we designed a *proportional–integral–derivative* (PID) controller [43, 45] as a stateful autoscaler that uses the historical autoscaling actions and current run-time measurements to make the next horizontal autoscaling decisions and keep the microservice "stable." Additionally, by taking into account the dependencies between different microservices, *SHOWAR* prioritizes the dependee microservices[1] to prevent unnecessary autoscaling actions and low resource utilization.

In addition to its use of autoscalers to determine resources for the microservices, *SHOWAR* aims to bridge optimal resource allocation and efficient scheduling of microservices. That is, once the optimal size for a microservice is determined, *SHOWAR* assists the cluster scheduler to schedule microservice for better end-to-end performance. In particular, to prevent resource contention and manage noisy-neighbor effects on microservice performance, *SHOWAR* uses estimated correlations of historical resource usage between different microservices to generate rules for the Kubernetes scheduler. These rules may, e.g., suggest the scheduler to co-locate (scheduling affinity) the microservices that have negative correlation of a certain resource type, or otherwise distribute them (scheduling anti-affinity).

We evaluate *SHOWAR* by deploying a variety of interactive microservice applications on a cluster of virtual machines on the Amazon Web Services (AWS) public cloud. We compare the performance of *SHOWAR* against two state of the art autoscaling systems: A moving window version of Google Autopilot [57] and the default Kubernetes autoscalers [27]. Using real-world production workloads, our results show that *SHOWAR* outperforms these baselines in terms of efficient resource allocation and tail distribution of end-to-end request latency. In particular, *SHOWAR* on average improves the resource allocation by up to 22%, which can directly translate to 22% in total savings of cluster-related costs, while improving the 99th percentile end-to-end user request latency by 20%.

In summary, we make the following contributions:

- *Vertical and Horizontal Autoscaler Frameworks*: We present the design of framework for both vertical and horizontal autoscalers for microservices that aim to improve the resource allocation efficiency.
- *Scheduling Affinity and Anti-Affinity Rules*: We bridge the gap between right sizing the microservices for resource efficiency and efficient microservices scheduling by generating scheduling affinity and anti-affinity rules to assist the scheduler for better microservices placement and performance.
- *Implementation and Evaluation*: We implement the designed frameworks and mechanisms creating a system called *SHOWAR* that works on top of the Kubernetes container orchestrator platform. Using real-world production workloads and real-world microservices we demonstrate the efficiency of *SHOWAR* compared to the state of the art baselines.

[1] i.e., microservices that others depend on

2 Background and Related Work

2.1 Background

In microservice architecture, the application service is broken-down into different individual microservices that work together to process a request. Figure 1 shows the Social Network application [38] as an example of microservices architecture. Each microservice is itself a containerized application that communicates with the other microservices using standard HTTP API requests or RPC calls. A user request arrives at the front-end layer of the application, and depending on the type of the request, it will span a subset of microservices (from front-end tier, middle tier, and back-end tier) to serve that request in the fashion of a directed acyclic graph (DAG). As can be seen in Figure 1, the microservices form a kind of dependency graph, moving from the front-end tier to the back-end tier, where some of the microservices obviously depend on others. This dependency graph information is used by *SHOWAR* to make better autoscaling decisions (see subsection 3.2).

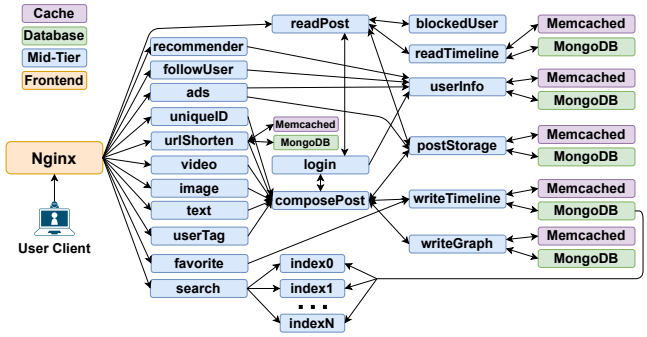

Figure 1. Social Network application as an example of microservices architecture

The increase in the popularity of microservices architecture has led to many industrial standards including container orchestrator tools for deploying microservices applications [2, 3, 60]. In this work, we use Kubernetes (k8s) as our container orchestration platform and build *SHOWAR* on top of it. In k8s, each microservice consists of at least one *Pod*, where each Pod consists of at least one application container, where a Pod is the smallest entity that k8s schedules. For better performance and availability, a microservice might have multiple replicated instances which we call service replicas or simply replicas herein. The size of each Pod in terms of compute resources such as CPU and Memory, and the number of replicas of it, are initially set (e.g., by the application owner, developer or user), and thereafter, the Kubernetes

autoscalers adjust the size of Pods. There are two types of autoscalers: The *Horizontal Autoscaler* which determines the number of replicas for each Pod, and the *Vertical Autoscaler* which determines the Pods' compute resources such as the amount of CPU and Memory. The autoscalers base their decisions on the historical resource usage of each Pod.

Deploying microservices requires the application owner to specify the IT resource requirements, including CPU, memory and disk, for its Pod's containers. Setting fixed values for each resource requirement of an application may be wasteful as the application's usage pattern changes over time. On the one hand, worst-case provisioning results in underutilized compute resources (wasted from the application owner's point-of-view) which is not cost-effective. On the other hand, under-provisioning the resources for the application could lead to throttled CPU or, worse, out-of-memory (OOM) errors which adversely affect end-to-end workload performance and hence user experience. In such situations, vertical autoscaling helps in finding the correct resource requirements for the application as workload and the usage patterns change over time, while meeting the application owner's service-level performance objectives (SLOs). By dynamically achieving optimal resource allocations for the application, an efficient vertical autoscaler a) minimizes the *resource usage slack* (i.e., slack = limit - usage) caused by over-provisioning and b) minimizes the number of OOM errors and the amount of time that the CPU is throttled due to under-provisioning.

Horizontal autoscaling refers to dynamically specifying the number of replicas for each microservice. As the load for a microservice increases, in order to prevent long backlogs in its request queues, horizontal autoscaling would increase the number of microservice replicas to maintain good responsiveness. In addition, in some situations, vertical autoscaling cannot increase the size of a service replica beyond a threshold. These thresholds can be the size of the largest virtual machine in the cluster, or the size beyond which the microservice does not benefit due to the lack of parallelism in the requests. Therefore, in such situations, one would use horizontal autoscaling to increase the number of replicas for the microservices.

In addition to CPU and Memory usage data, *SHOWAR* uses the extended Berkeley Packet Filtering (eBPF) [6] metrics data for its horizontal autoscaling decisions. eBPF is a recent Linux kernel technology that enables running secure and low-overhead programs at the kernel level to collect accurate metrics from the kernel level events such as CPU scheduler decision events, memory allocation events, and events of packets in the kernel's networking stack. It has been widely used for microservice observability toward a wide range of purposes such as performance improvements, profiling and tracing, load balancing, network monitoring and security [5, 7–9].

2.2 Related Work

Here, we give a brief overview of the state of the art related work on autoscaling and resource management for microservices which we build upon and use as our baselines.

Autoscaling Microservices. Autoscaling has been studied in the context of public cloud resources [13–15, 50], including different types of workloads [33, 46] and microservices [27, 40, 57]. The autoscaling framework for microservices which is widely used by practitioners in industry are those of Kubernetes [27] and Google Autopilot autoscaler [57].

For vertical autoscaling, the Kubernetes vertical autoscaler uses the historical resource usage of both CPU and Memory and calculates the 90^{th} percentile (P90) of the aggregated resource usage across the replicas of each microservice for the last (sliding) window of N (a system parameter) samples. It then sets the resource (CPU and Memory) limits for the microservice for the next time window to be P90×(1.15). The extra 15% is used as a safety margin to avoid under-provisioning resources for the microservice. Google Autopilot's vertical autoscaler takes the same approach with a slight modification. In particular, for the Memory resource, instead of the P90, it uses the *maximum* (max) Memory usage in the last N samples as the limit for the next window time. But for the CPU resource, it uses P95×1.15 (not P90) for the next time window. In *SHOWAR*'s vertical autoscaler design, instead of using high percentiles in this way, we use the empirical variance in resource usage for improved resource allocation to the microservices (see subsection 3.1).

For horizontal autoscaling, both Kubernetes and the Google Autopilot take the same approach. The application developer/user sets a target CPU utilization T^* for each microservice[2]. At any time t, for each microservice M, the number of replicas R_M is calculated as follows:

$$R_M = \frac{\sum_r \text{P95}_r}{T^*},\ r \in \{M\text{'s current replicas}\}$$

where P95_r is the 95^{th} percentile of the CPU usage of the replica r for the microservice M in the last N samples. In *SHOWAR*'s horizontal autoscaler design, we take a totally different approach and instead of using CPU utilization as an autoscaling metric, we use the CPU scheduler's eBPF metrics to design a more accurate and stateful autoscaler (see subsection 3.2). As another alternative to the sliding window approach, the Google Autopilot vertical autoscaler also employs an ensemble of machine learning models for its vertical autoscaler. The models try to optimize a cost function of autoscaling actions (e.g. under- and over-provisioning) and the model with the lowest cost is picked to perform a vertical autoscaling action on each microservice.

Autoscaling to meet SLOs. Another line of prior work, which is orthogonal to autoscalers such as *SHOWAR*, utilizes autoscaling actions as a means to prevent service level

[2]Usually $T^* \in [60\%, 75\%]$.

objective (SLO) violations due to microservice straggling [37, 47, 55, 58, 62]. For example, FIRM [55] is a machine learning (ML) based system for predicting and mitigating the SLO violations in microservices. It first localizes the SLO violation to determine which microservice in the dependency graph is contributing to the SLO violation the most, then using a pre-trained reinforcement learning (RL) agent, FIRM performs proactive autoscaling actions on different resource types (CPU, Memory, LLC etc.) to mitigate the SLO violation. Sage [37] is a supervised ML based system designed to mitigate performance problems in microservices. In particular, it models the dependencies between the microservices using causal Bayesian networks to identify the root cause of SLO violations and then performs autoscaling actions to mitigate the SLO violation. The fundamental limitation of this line of work is that they use pre-trained machine learning models that cannot be easily adapted to the changes in the workload (i.e. workload shift), and as a result they can take poorly chosen actions. Though some of these works propose incremental retraining, the cost and the time it takes to retrain the models could be a hurdle towards adapting sufficiently quick to the changes in the workload. In contrast, *SHOWAR*'s autoscaler controllers adapt quickly to the workload changes since they use the recent resource usage statistics and workload changes as signals for their autoscaling actions.

3 *SHOWAR* Design

In designing *SHOWAR* we aim to bridge between vertical pod autoscaling, horizontal pod autoscaling, and pod scheduling and placement for efficient resource usage and good end-to-end performance. In the following, we describe the design of each part and discuss how they work in tandem.

3.1 Vertical Autoscaler

The state of the art of vertical autoscaling systems, specifically Kubernetes vertical autoscaler (and similarly Google Autopilot) [22, 57], take a conservative approach to set the CPU and memory requirements for microservices. In particular, by monitoring the historical resource (both CPU and Memory) usage in a past window of time (e.g. few minutes to several days), and then setting the resource allocation for the next window of time to be some percentile π of the usage and a safety margin α: $\pi(1+\alpha)$ where typically π is between 90^{th} and 99^{th} percentile and α is between 0.10 and 0.2. While this approach of taking a high percentile of the historical usage and adding some safety margin to it addresses the under-provisioning problem and minimizes the number of OOM errors as well as the time that the CPU is throttled, it still leaves the problem of underutilized and wasted resources which is not cost-effective for the application owner.

In *SHOWAR*'s vertical autoscaler we take a simpler approach while addressing both resource under- and over-provisioning and their implications. Particularly, *SHOWAR* uses a standard "three-sigma" rule-of-thumb to set the resource allocations of the microservices. To that end, the usage statistics of each resource type (CPU or memory) from the last window of duration W seconds (collected every second, see section 4) are used to recursively compute the mean μ and its variance σ^2 over that window. Then $s = \mu + 3\sigma$ is the estimated amount of that specific resource type currently needed by application. The amount s is evaluated every T seconds where $T \ll |W|$ and, if it has changed substantially (say more than 15%) since last evaluation, then the allocation is updated.

SHOWAR's vertical autoscaler simply prevents both under- and over-provisioning by taking the usage pattern's variance. Compared to the prior work (i.e. using $\pi(1+\alpha)$) where a fixed amount of resources is added to to its tail percentile usage, when there is a high variance in the resource usage, *SHOWAR* provisions the resource accordingly and hence prevents under-provisioning and performance degradation. In addition, when there is a low variance in the resource usage, *SHOWAR* does not over-provision and hence prevents over-provisioning and wasted resources.

To reiterate, while both $\mu + 3\sigma$ and $\pi(1+\alpha)$ have clear statistical interpretation, the choice of using 3σ gives a more accurate view of the "spread" of the distribution about the mean. In particular, if the variance is very small, then the distribution is almost constant which is separately useful information about the resource usage of the Pods. However, in $\pi(1+\alpha)$ method, when the variance is very small, the tail percentiles do not convey useful information. Additionally, the choice of the safety-margin hyperparameter α may be arbitrary which can result in poor resource utilization or more OOMs if it is not specified properly.

3.2 Horizontal Autoscaler

While being widely used in production, the state of the art of horizontal autoscaling systems [21, 57] suffer from a few shortcomings in their design. First, they *proportionally* react to the current autoscaling metric (i.e. CPU utilization) measurement (compared to the target metric value, i.e. observation minus target), and as a result they increase or decrease the number of service replicas to reach to the target metric value in a single shot and in a stateless fashion. This becomes problematic and inefficient when there are bursts and fluctuations in the load, which can result in extreme over-provisioning (so not cost effective) or under-provisioning (so performance degradation and poor quality of service), cf. section 5. To address such problems, some autoscaling systems introduce the notion of a cool-down period in which no autoscaling actions are performed for a period of time right after the most recent action[3]. While this can mitigate the number of abrupt autoscaling actions, a transient spike in the load can fool the autoscaling system to perform an unnecessary autoscaling action while missing the required

[3] Alternatively, one can add "hysteresis" to autoscaling rules to prevent too-frequent control actions.

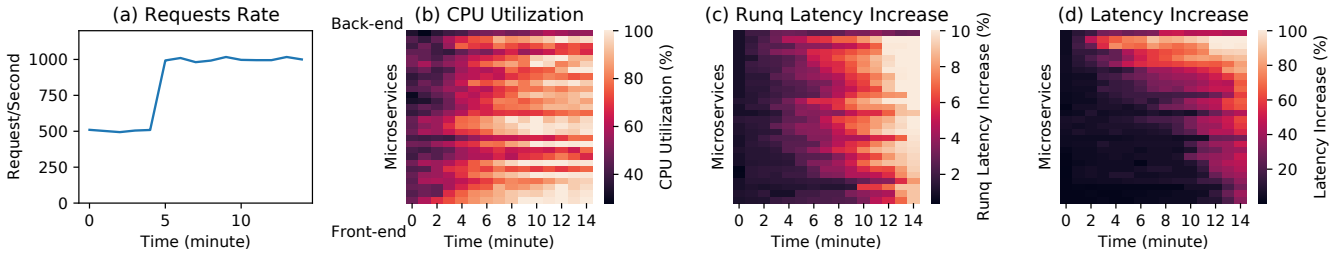

Figure 2. Heatmap of latency propagation from back-end microservices to the front-end microservices when there is a change in the workload. Lighter color denotes higher value (consult the individual color bars for exact value range). (a) Request arrival rate. The arrival rate increases at $t = 5m$ to make higher load on the application. (b) CPU utilization of each microservice from back-end tier (top) to front-end (bottom) tier. As the load increases, the CPU utilization of microservices in all tires increases. (c) The increase in *runq latency* of each microservice. As the load increases, the *runq latency* increases over time from back-end microservices to the front-end microservices. (d) The increase in request latency of each microservices. As the load increases, the back-end microservices face higher increase in their latency and propagates to the front-end microservices over time.

autoscaling decision for the next time-window because of the cool-down period.

Second, these systems usually do not take the request's dependency graph for its microservices into the account and treat each microservice independently. Prior works have shown that resource allocation and autoscaling for microservices without considering correlations among dependent microservices results in inefficient resource allocation and does not necessarily help in coping with the load changes and maintaining a good performance [39, 48, 55]. To illustrate, we deployed the *Social Network* application to see how different microservices react to changes in the load by monitoring different performance metrics. As can be seen in Figure 2, as the load increases at time $t = 5m$ (in Figure 2a), the back-end microservices start experiencing high request 99^{th} tail latency (in Figure 2d) compared to normal operation. The high tail latency then progressively propagates from the back-end microservices to the middle tier and finally to the front-end microservice as time passes. An efficient horizontal autoscaling system would autoscale the back-end microservices first to prevent high tail latency propagation and possibly avoid unnecessary autoscaling for the front-end microservices.

Finally, previous approaches typically use CPU utilization as a metric for autoscaling decisions by striving to maintain a target CPU utilization across all the microservices. However, it's well-known that CPU utilization is not the most effective metric for autoscaling and resource allocation [23, 34, 36, 41, 42, 49, 53]. As it can be seen in Figure 2b, as load increases, the CPU utilization increases for almost all the microservices, while the tail latency of front-end microservices does not always increase as CPU utilization increases. That is, the high CPU utilization does not always translate into high tail latency increase as there are microservices that have high CPU utilization but do not experience high tail latency. As a result, the state of the art autoscaling systems would perform unnecessary autoscaling actions for the microservices (front-end microservices for example) as they are not contributing to the request latency in a very significant way, cf. section 5.

In *SHOWAR*'s horizontal autoscaling design, we aim to address the shortcomings that we mentioned above. We propose to use a basic framework of control theory to design a stateful horizontal autoscaling system that maintains stability while meeting performance requirements as expressed by the metrics of SLOs.

The Autoscaling Controller. As mentioned above, reacting just proportionally to the observed autoscaling metric error,

$$e = \text{observation - target},$$

i.e., proportional control, is not enough and can result in poor and inefficient autoscaling decisions and instability owing to abrupt changes in the number of replicas. To improve this, we appeal to a more sophisticated controller: A *proportional–integral–derivative* (*PID*) controller [44, 52] which effects control based on a linear combination of three terms at time t:

$$u(t) = k_P e(t) + k_I \int_{t-w}^{t} e(\tau)d\tau + k_D \frac{d}{dt} e(t),$$

where, e.g., integration window can be set to $w = t$ in particular, and k_P, k_I, and k_D are the coefficients for the proportional (reacting to changes), integral (the memory for the past state), and derivative (predicting the future state) terms respectively [43, 45]. These parameters are further discussed below.

Metric. As mentioned before, CPU utilization is not always the most effective basis for horizontal autoscaling. We propose to use more meaningful and low-level CPU performance metrics. In particular, we use eBPF Linux scheduler *runq latency* metric [24] which represents the time between when a thread is runnable, and the time when it acquires the CPU and is running. The longer the *runq latency*, the higher CPU

contention among the threads and vice versa, therefore, as shown by prior work [48], this metric is a suitable choice for autoscaling purposes. In addition, from a control perspective, it is a metric that changes fast enough in response to an autoscaling action (i.e changing the number of replicas), as such stabilizes the system faster. Furthermore, this metric captures both changes in the incoming load for the service as well as the input signals (i.e., changing the number of replicas) from the autoscaling system controller. *Runq latency* is represented as a histogram of the latencies that threads experience. As a target point for the autoscaling system controller, we use, e.g., $\pi = 95$ percentile value of this histogram. The controller performs the autoscaling actions so that such target values for *runq latency* are met. To illustrate the effectiveness of *runq latency*, Figure 2c depicts the *runq latency* increase in microservices. As it can be seen, the *runq latency* shows a similar behavior to the request tail latency and it progressively increases and propagates from the back-end microservices to the front-end microservices. Unlike CPU utilization, the high *runq latency* is highly correlated with the high request tail latency for each individual microservice which shows that the *runq latency* can be used as a suitable metric for horizontal autoscaling to prevent request latency increase. Intuitively, the reason that *runq latency* is superior to CPU utilization is that it indicates how the application threads are *competing* for CPU resources and hence the need for more (or less) CPU resources [23].

Note that even though *runq latency* is a per thread metric, it is still applicable to single-threaded applications. This is because even in a single-threaded application, the thread can be interrupted by the OS and hence an increase in its *runq latency*. In addition, when the this metric is low for an application, and horizontal autoscaler cannot be triggered, the vertical autoscaler of *SHOWAR* can proceed and allocate resources for the Pod or trigger the horizontal autoscaler to scale the pod horizontally (see subsection 3.3 for more details).

High-level application latency SLOs are business decisions and generally may not directly translate to *runq latency*. However, as shown in Figure 2, the lower *runq latency* results in lower application latency and vice versa. As such, specifying a target value for the *runq latency* here can result in meeting the target application latency SLOs. In current *SHOWAR*'s design and deployment, the user (i.e. the application owner) has to specify a target runq latency value as part of *SHOWAR*'s configuration. Since *runq latency* is an OS scheduler metric, the value for the target *runq latency* should not be higher than a few orders of magnitude of CPU scheduler time-quanta (where a quantum is configurable).

The Transfer Function The transfer function in our autoscaling case is simple and has the property that: If *runq latency* exceeds the target value, then the autoscaling system has to scale out and increase the number of replicas. Also, if *runq latency* is below the target value, then the autoscaling system has to scale in and decrease the number of replicas. The overall procedure is shown in Algorithm 1. To prevent performing too many autoscaling actions in response to fast changes as well as transient burstiness in the *runq latency* metric, we set a configurable bound α percent (20% by default) around the target value as a buffer and no autoscaling action is performed (i.e. NO-OP). The amount of increase or decrease in case of autoscaling is a configurable β percent (10% by default) of the number of current replicas of the microservice or is 1 if β is less than 1[4].

Algorithm 1: Autoscaling Transfer Function

```
M : microservice;
PID: pid controller for M;
R_M: Number of replicas for M;
α: PID's action bound;
β: Replica change step coefficient;
while True do
    runqlatency = runq_sample_histogram(M);
    observation = P95(runqlatency);
    output = PID.output(observation);
    if output > target * (1 + α/2) then
        R_M = R_M + max(1, R_M * β);
    else if output < target * (1 − α/2) then
        R_M = R_M − max(1, R_M * β);
    else
        NO − OP ;
    end
end
```

Tuning The Autoscaling Controller. Tuning the autoscaling PID controller refers to determining the values for the coefficients k [43, 44, 52]. Different values for the coefficients can affect the performance of the controller in terms of speed (responsiveness), stability and accuracy. In particular, increasing k_P leads to an increase in the speed of the controller (to reach to a stable state), however high k_P values may correspond to instability which is a main problem of prior works relying only on proportional autoscaling control, e.g., [21, 57]. Increasing k_I increases the speed of the controller as well and may result in instability, but increasing k_I will lower the controller's noise (variation and fluctuations) and steady-state errors. Finally, increasing k_D increases the speed of the controller (to reach steady-state) as well as the potential for instability while amplifying the controller's noise profoundly. As a result, given the effect of the coefficients values on the controller's speed, stability and noise, a workload-aware and adaptive tuning approach is required.

Instead of using traditional and standard PID tuning methods [43, 44], we propose to use the following adaptive method

[4]For most of the microservices at our evaluation scale, 10% of replicas had $\beta < 1$, cf. section 5.

mainly to cope with the variations in the incoming workload (i.e. workload shift). This is because failing to cope with the workload changes can result in poor resource efficiency or worse, service downtime due to insufficient resource allocations [11].

Initially, the controller starts with equal values for the coefficients. Subsequently, the coefficients are adaptively and incrementally self-tuned based on monitored workload performance and controller state. In particular, if the current metric value (especially *runq latency*) is far from the target metric value, k_P and k_I are increased in each iteration to improve stability as well as the speed at which the target metric value is reached. Also, if fluctuations in the metric value are observed (referred as noise in the controller), k_D is decreased gradually to reduce the noise introduced by the workload's burstiness.

Autoscaling Approach. As explained above, a fundamental problem with state of the art horizontal autoscaling systems is that they can perform unnecessary autoscaling actions that can result in poor resource utilization. To address this problem, in *SHOWAR*'s horizontal autoscaling design we take into the account two knobs: a) the sorted absolute values of the controllers' output for each microservice (cf. the "one for each design" case), and b) the (topologically sorted) dependency graph among the microservices. We propose two architectures for the horizontal autoscaling controller system:

- **One For All:** In this design, a single controller is responsible for autoscaling all of the microservice types. That is, at every autoscaling decision, all the microservices are scaled (up or down) at once depending on the average of current metric value observation across all of the microservices. While this approach benefits from the PID controller, it does not take into the account the microservices dependency graph of the microservices.
- **One For Each:** In this design, a controller is responsible for each microservice. Each controller monitors the autoscaling metric (runq latency) of its corresponding microservice and performs the autoscaling action for that microservice (according to Algorithm 1), in a coordinated fashion as follows. The absolute value of controllers' outputs are sorted and those with the highest values (greatest scaling need) are prioritized. For equal controllers' outputs, we then take into the account the dependency graph of microservices, and prioritize the back-end services over the dependent front-end services (after a topological sort of the graph). In our benchmarks, we observe the back-end services typically are the ones with the highest controller's output values as well (see section 5). Motivated by the observations in Figure 2, once a microservice's controller has performed an autoscaling action, the controllers for all of its dependent microservices are postponed until the autoscaling is done (i.e. new replicas are added, or some of the replicas are removed) and then the dependent microservices' controllers try to perform autoscaling. This is because, in most of the cases, autoscaling the dependee microservice eliminates the need for autoscaling the dependent microservices and hence the latency increase propagation observed in Figure 2.

3.3 Tandem Vertical and Horizontal Autoscalers

The recommended approach for deploying vertical and horizontal autoscalers in state-of-the-art[5] platforms such as Kubernetes [21, 22] is to only deploy one autoscalers at a time to avoid interference from others. That is, in case of choosing the vertical autoscaler, the developers would set a fixed number for the number of replicas (instances) for each microservice and the vertical autoscaler will scale up or down the size of Pods running the microservice. On the other hand, if a horizontal autoscaler is chosen, the developers would, for each microservice, set a fixed size (CPU and memory), and the horizontal autoscaler will scale up or down the number of replicas for each microservice.

SHOWAR benefits from both vertical and horizontal autoscaling by allowing deploying them in tandem. First, we prioritize any vertical autoscaling decision over any horizontal autoscaling decision. We do this because, in case of memory autoscaling for example, if there is insufficient memory for the Pod, the application encounters an out of memory (OOM) error and stops the execution regardless of its replicas count, as such, the horizontal autoscaler cannot address the problem. Therefore, before a horizontal autoscaling controller acts, it first checks a shared channel to see if a vertical autoscaling is in progress for that microservice and, if so, it will not proceed. Similarly, before a vertical Pod autoscaler acts, it sends a message over the shared channel notifying the horizontal autoscaler and then performs its action.

Second, according to Google Cloud Platforms' Kubernetes best practices, due to lack of parallelism, it's recommended that for most of the workloads, no more than one core (i.e. $1000m$ core in kubernetes currency) is needed for each Pod [20]. We use this recommendation and incorporate it into the *SHOWAR*'s vertical autoscaler design. That is, if the vertical autoscaler decision is to set more than one core for a Pod, it signals the horizontal autoscaler through a shared channel instead, and will not proceed with the vertical autoscaling action.

These two mechanisms allow the vertical and horizontal autoscalers to be deployed at the same time and work in tandem for efficient resource allocation while maintaining the target performance goals.

[5]Google Autopilot allows deploying both vertical and horizontal autoscalers at the same time.

3.4 (More) Efficient Scheduling

Again, the general aims of vertical and horizontal autoscaling are twofold: a) efficient resource usage and allocation while b) maintaining a good performance for the microservices. However, in addition to autoscaling, another factor that helps in efficient resource usage and meeting performance goals for the microservices is scheduling and placement of the microservice Pods on the nodes (VMs) in the cluster. In particular, the scheduler would place as many Pods as possible on a node to improve the resource usage once the Pods are right-sized both vertically and horizontally by the autoscalers. On the other hand, "tight" (efficient) placement of the Pods into nodes can result in resource contention (due to noisy neighbor effect) hence degrading the performance of Pods as well the microservice as a whole.

To bridge the gap between scheduling and autoscaling, and to improve the resource efficiency as well as the performance of the microservices even further, *SHOWAR* provides "hints" (or rules) for the Kubernetes scheduler's by generating inter-pod affinity and anti-affinity rules. An affinity of service S_2 for service S_1 implies that the scheduler will always (or preferably) try to schedule the Pods of service S_1 on the nodes which Pods of service S_2 reside on. Similarly, an anti-affinity of service S_2 for service S_1 implies that the scheduler will never (or preferably not) do this.

In doing so, *SHOWAR* monitors and uses the historical (i.e. last (configurable) window of time) CPU, memory, and network (both out and in) usage of microservices and calculates the Paerson correlation coefficient [32] between each pair of microservices' usage pattern: Given the distribution of CPU (or memory or network I/O) usage of two microservice types X and Y, the correlation coefficient ρ between X and Y is:

$$\rho_{XY} = \frac{\sum_{(x,y)\in S} \sum (x-\mu_X)(y-\mu_Y)}{\sigma_x \sigma_y} \in [-1, 1],$$

where μ_X and σ_X are the (historical) sample mean and sample variance of X, respectively, each computed by standard recursive means.

For two microservices S_1 and S_2, the higher positive correlation in the usage pattern of a resource (say CPU or memory), the higher resource contention for that resource between them. Similarly, the lower the negative correlation, the lower the contention between the two services for that resource. This is the simple basis of *SHOWAR*'s affinity and anti-affinity rules for the compute resources such as CPU, memory, and network I/O.

Generating Affinity and Anti-Affinity Rules. Specifically, the mechanism for generating affinity and anti-affinity rules for the scheduler based on the correlation coefficient for each resource type is as follow:

- **CPU and Network:** For any pair of microservices S_1 and S_2, if they have a *strongly negative* correlation in their CPU and network I/O usage pattern (i.e. $\rho_{S_1S_2} \leq -0.8$) *SHOWAR* generates an *affinity* rule of S_2 for S_1 for the scheduler. This is because CPU and network bandwidth are resources that can be shared and throttled, and, as such, even if the usage pattern of the two service change and the negative correlation does not hold, the microservices may be able to effectively share such resources.
- **Memory:** If any pair of microservices S_1 and S_2 have a *strongly positive* correlation in their memory usage pattern (e.g. $\rho_{S_1S_2} \geq 0.8$) *SHOWAR* generates an *anti-affinity* rule for S_1 and S_2 for the scheduler. This is because when the memory usage pattern of two services is strongly correlated, given the limited physical memory bandwidth on the node, the microservices can suffer from insufficient memory bandwidth.

Note that an anti-affinity rule has a symmetric property that can completely prevent the scheduler from scheduling two strongly memory correlated microservices. In other words, if there is an anti-affinity rule of S_2 for S_1, unlike an affinity rule, when scheduling Pods of microservice S_1, the scheduler not only checks the presence of Pods of microservice S_2, but also, when scheduling Pods of microservice S_2, checks the presence of Pods of microservice S_1 to not co-schedule/locate them on the same node (even though there is no affinity rule set of service S_2). As a result, to not make any scheduling conflicts for the scheduler, *SHOWAR* generates at most one affinity or anti-affinity rule for each microservice. That is, each microservice participates in at most one affinity or anti-affinity rule at any point in time.

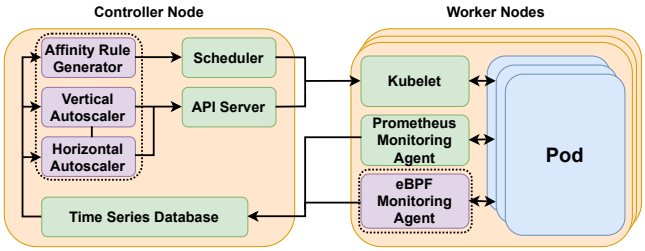

Figure 3. *SHOWAR* Architecture Overview. The resource usage logs as well as the eBPF metrics are collected using their corresponding agents on each node and are aggregated into the time series database. *SHOWAR* uses the collected metrics to make autoscaling decisions as well as scheduling affinity and anti-affinity rules by communicating with the Kubernetes API server and its scheduler respectively.

4 *SHOWAR* Implementation

SHOWAR is implemented in GoLang as a cloud native programming language and consists of a set of modules plugged into the state of the art container orchestrator Kubernetes. Figure 3 depicts a high-level overview of the architecture and how *SHOWAR* is interacting with the kubernetes scheduler and its API server. *SHOWAR* is deployed as a service on

the controller node and interacts with the kubernetes API server and its scheduler for autoscaling actions as well as applying the generated affinity and anti-affinity rules for the microservices.

Monitoring Agents. The monitoring and logging data are the most essential part of any application deployment. The monitoring data are used for observability, health check and autoscaling. We use the state of the art monitoring and metric collection tool Prometheus [25] to collect different metrics from nodes and containers. Prometheus launches a monitoring agent on each node in the cluster to collect the container metrics such as CPU usage, Memory usage, Network bandwidth usage, etc. The agents are configured to collect and report the metrics every second (One second is the minimum period that Prometheus agents can collect the metrics. To obtain as many as data points possible, we collected data every second.). Prometheus comes with a time series database where the agents store the collected metric. In addition, a query language is provided to query the time series database which is used by the other modules to utilize the collected metrics.

In addition to the Prometheus standard metric collection agents, we have developed an eBPF program that is deployed as monitoring agent on every node in the cluster to collect the *runq latency* metric used by the horizontal autoscaler. This metric is a histogram of latencies that the CPU threads in each pod experience before acquiring the CPU. The program collects a histogram of *runq latencies* every 1 second and stores it in the Prometheus time series database.

The Vertical Autoscaler. The vertical autoscaler is a simple loop that takes place every minute[6]. That is every minute, it evaluates $s_r = \mu_r + 3 * \sigma_r$ over a window of the previous 5 minutes for each resource type r (CPU and memory) and if the value of s has changed by more than 15%, it updates the resource requirements of the service to be s. Another condition that triggers the vertical autoscaler is when a microservice reports an OOM error. Before applying the new resource requirements of the microservice, the vertical autoscaler sends a message over a shared channel to the horizontal autoscaler to not proceed with any horizontal autoscaling action as vertical autoscaling actions are prioritized over horizontal autoscaling. The vertical autoscaler also does not proceed with an autoscaling action for a microservice if the amount of CPU for that microservice is more than one CPU core (i.e. $s_{CPU} > 1000m$), in that case, it sends a message over another shared channel to the horizontal autoscaler to trigger a horizontal autoscaling action.

The Horizontal Autoscaler. As described in subsection 3.2, at its core, the horizontal autoscaler is a PID controller that aims to keep each microservice stable. That is, for a given target *runq latency*, it performs horizontal autoscaling actions for that microservice such that it always has a *runq latency* of the target value. The controllers make decision every 1 minute which the eBPF program collects 60 instances of the metric histogram (1 every second). For each histogram, the 95^{th} percentile is picked and the controller uses the average of these 60 data points as its current observation (a.k.a measurement) to perform its controlling action. Each horizontal scaling action adds or removes at least 1 or a configurable percentage (10% by default) of current number of replicas of the microservice for scaling in and out respectively.

Recalling from section 3, the initial values for the PID control parameters are taken to be $k_P = k_I = k_D = 1/3$ (each parameter constrained to be $\in [0, 10]$). Incremental changes to these parameters is 10% (we found experimentally that 10% gives very good performance). Fluctuations in the controller's output, which are a basis to make such changes, is measured using the previous $N = 10$ samples. Also, the "speed" of the controller is measured as the number of iterations required to reach the interval $[\text{target}(1-\alpha), \text{target}(1+\alpha)]$ for $\alpha = 10\%$.

The Affinity Rule Generator. *SHOWAR*'s affinity rule generator uses the CPU, memory, and network utilizations every 5 minutes which is a vector consisting of 300 data points (each data point is the average over the microservice replicas) to compute the correlation coefficient of different resource types between every pair of microservices. Too eliminate the weak or no correlation instances, any value in $[-0.8, +0.8]$ is dropped. The other strongly negative and strongly positive correlated microservices are used to generate the affinity and anti-affinity rules as explained in subsection 3.4. The resource usage patterns can change as the workload changes (also known as workload shift), so if a strongly negative or positive correlation change by more than 20% (configurable) in a subsequent 5 minutes window of time, *SHOWAR* revokes the affinity (or anti-affinity) rule for that pair of microservices.

***SHOWAR*'s overhead.** Note that *SHOWAR* is built as a controller for the Kubernetes which is highly pluggable for autoscalers and other types of controllers [30]. In addition, *SHOWAR* uses the commonly used Kubernetes monitoring agents (e.g. Prometheus [25]) and one custom eBPF metric monitoring agent. As such, compared to the default Kubernetes autoscalers, *SHOWAR* does not introduce any additional overhead. Furthermore, the autoscalers are scheduled on the controller node and do not share resources with the application Pods which are scheduled on the worker nodes.

5 Evaluation

5.1 Experimental Setup

Applications. We evaluated *SHOWAR* using 3 interactive microservice applications: a) *Social Network* from DeathStarBench [38], an application consisting 36 microservices in

[6]The decision frequency is configurable. We chose a frequency of once per minute since one minute is about the minimum amount of time over which an ample amount of metrics data are available to well inform an autoscaling decision.

which users can follow others, compose posts, and read and interact with others' posts; b) *Train-Ticket* [64], an application consisting 41 microservices which allows its users to reserve online tickets and make payments; and c) Google Cloud Platform's *Online Boutique* [28], consisting of 10 microservices in which users can purchase online items through their online cart and make payments. Through our experimental evaluation, we observe that the results are consistent across all 3 application benchmarks, therefore due to space limits, we only report the results for the *Social Network* application. We set the target value for *runq latency* to $15ms$ which is $2.5x$ the Linux kernel *sysctl_sched_latency*[31] scheduler parameter.

Cluster Setup. All of our evaluations are performed on Amazon Web Services (AWS) Cloud in *us-east-1* region. We use $m5.xlarge$ VM instances each with 4 vCPU, 16 GB of memory and $\$0.192/hr$ price, running Ubuntu 18.04 LTS configured for supporting running eBPF programs. Unless otherwise mentioned, our cluster consists of 25 VM instances.

Workload and Load Generation. We use Wikipedia access traces [59] as our primary workload. It's a real-world trace of users interacting with the Wikipedia website consisting of traffic patterns including periods of Poisson arrival times, short-term burstiness, and diurnal level-shifts. Since the microservices that we are evaluating are user-facing applications, the workload has to reflect realistic user behavior. As such, the Wikipedia access trace is a good fit for our evaluation. We use locust [26] as our workload generator in a distributed fashion. The locust clients reside on different VM instances than the main cluster that is hosting the application.

Baselines. We compare *SHOWAR*'s performance with two main baselines: a) Kubernetes default autoscalers [27] and b) Google Autopilot [57]. We implemented a version of Google Autopilot moving-window vertical autoscaler as described in [57] – its ML-based version is not used as it has not yet been open-sourced and not enough information has been disclosed to re-implement it.

5.2 Vertical Autoscaling

We first evaluate the effectiveness of *SHOWAR*'s vertical autoscaler (horizontal autoscalers are disabled) in reducing the relative memory slack. Our resource of interest here is memory because an insufficient allocation of memory for a service can result in out-of-memory errors which affect service availability, but CPU can be throttled and keeps the service available at the cost of degraded performance. We use a one-hour long workload from the Wikipedia access trace shown in Figure 4 for our evaluation. We log the memory limit set by the vertical autoscaler for each microservice as well as the microservice's actual usage every 5 minutes to calculate its memory usage slack (i.e., slack = limit - usage). Figure 5*a* depicts the cumulative distribution function (CDF) of relative memory usage slack (i.e. $slack/limit$) across all

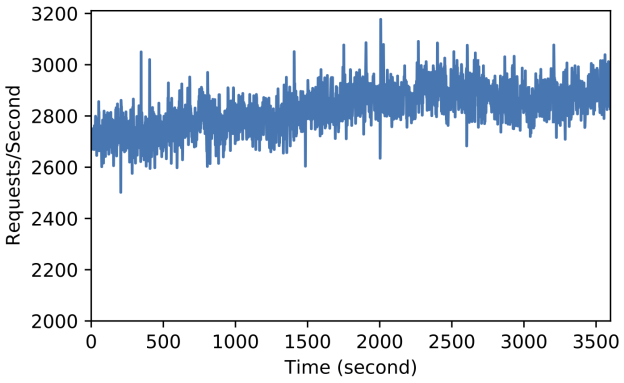

Figure 4. A one-hour long workload from Wikipedia access trace.

the microservices and their replicas in *Social Network application*. As can be seen, by embracing the variance of past resource usage (using three-sigma rule), *SHOWAR*'s vertical autoscaler is able to improve the memory usage slack compared to Autopilot's and Kubernetes' vertical autoscaler which use max (maximum) and P90 × 1.15 (15% more than the 90^{th} percentile) of past usage respectively. In particular, for 95% of the service instances, the relative memory usage slack is less than 46% compared to 63% and 66% for Kubernetes and Autopilot respectively. This 20% savings in memory usage slack can be utilized for scheduling more instances of services or using less VM resources in the cluster which will obviously reduce costs (see subsection 5.5). We also observe that Kubernetes outperforms Autopilot as it has a more aggressive approach in setting the limits (using P95 × 1.15 of past usage compared to the max).

While low memory or CPU usage slack can result in efficient and cost-effective resource allocation, it can however result in higher rates of OOMs or throttled CPU and hence degradation in service performance. Figure 5*b* shows the number of OOMs over the course of the experiment. As can be seen, while *SHOWAR* has comparable number of OOMs compared to Kubernetes, their aggressive approaches in memory scaling result in more OOMs compared to the Autopilot. In Figure 5*c* we depict the average CPU throttling (result of tight CPU slack) across the microservices during the course of the experiment. When the CPU usage of a Pod exceeds its allocated CPU resources, the container runtime (using *cgroups*) throttles the CPU share of the Pod. As can be seen, *SHOWAR* has a CPU throttling comparable to the baselines, because of the high fluctuation (variance) in the CPU usage of the microservices.

As can be seen in Figure 5, there is a natural trade-off between resource efficiency (i.e. lower slack) and stability. *SHOWAR* and Kubernetes result in better resource efficiency while resulting in higher number of OOMs (and throttled

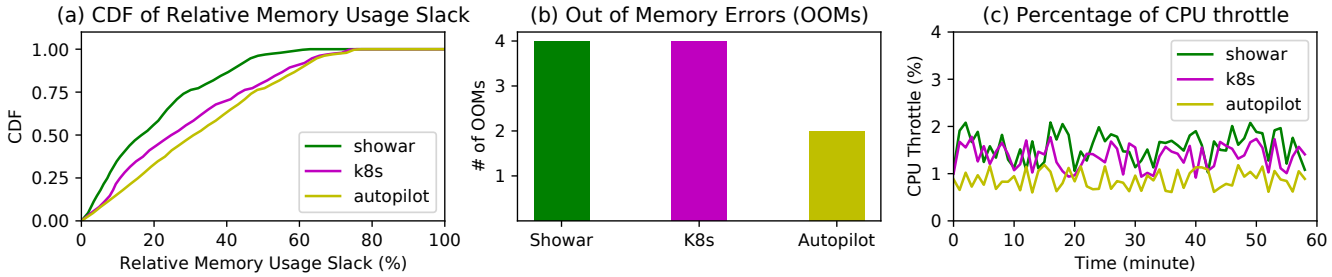

Figure 5. Vertical Autoscaling: (a) CDF of relative memory usage slack. (b) Number of Out of Memory (OOM) Errors. (c) Average CPU throttling across all the microservices.

CPU), while Autopilot results in higher slack and less number of OOMs. Depending on the objectives, one can tune *SHOWAR* and Kubernetes to achieve higher stability at the cost of higher resource usage slack. For example, in *SHOWAR*, instead of the 3σ term, one can use $k\sigma$ where $k > 3$ to allocate more resources for individual Pods and mitigate OOMs and CPU throttling.

5.3 Horizontal Autoscaling

Here, using the same workload from Figure 4 we evaluate the effectiveness of *SHOWAR*'s horizontal autoscaler (vertical autoscalers are disabled) in determining the right number of replicas for microservices. We compare both *SHOWAR*'s *One for Each* and *One for All* designs with Autopilot and Kubernetes horizontal autoscalers. Recall from subsection 2.2 that both Autopilot and Kubernetes use the same approach in horizontal autoscaling. We set the target CPU utilization for Autopilot and Kubernetes to 65% as it is commonly recommended.

Figure 6 depicts the cumulative distribution function of the number of replicas of microservices in *Social Network* application over the course of the experiment. We observe that both *SHOWAR*'s horizontal autoscalers outperform the Autopilot and Kubernetes horizontal autoscalers by allocating fewer replicas for the majority of microservices, which in turn can result in more efficient resource allocation and cost savings (see subsection 5.5). By having a tailored controller for each microservice, *SHOWAR*'s *One for Each* design also outperforms its *One for All*. This is because in the *One for All* design, a single controller tries to scale the microservices using a single target *runq latency* value and an averaged *runq latency* measurement across all the microservices which results in unnecessary scaling of microservices that do not have, high *runq latency*. In addition we observe that, as expected, Autopilot and Kubernetes have almost identical horizontal autoscaling decisions because they use the same method for horizontal autoscaling using the same target CPU utilization. The slight difference between the two comes from the variations in CPU utilization measurements during the experiments.

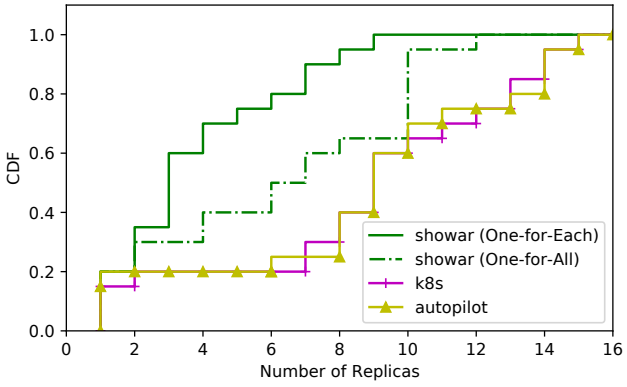

Figure 6. Horizontal Autoscaling: CDF of number of replicas across microservices.

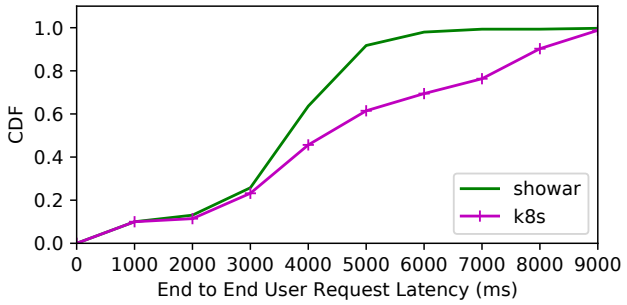

Figure 7. Affinity Rule Generator: CDF of user-experienced end-to-end P99 latency in presence of *SHOWAR* affinity rule generator for CPU, Memory, and Network I/O compared to Kubernetes default scheduler.

To reiterate, the effectiveness of *SHOWAR* is due to a) a stateful controller for the autoscaler and b) a better representative metric (i.e. *runq latency* instead of CPU utilization) for autoscaling decisions. The effect of being stateful and having memory of the past autoscaling actions can be seen in Figure 6 where *SHOWAR* allocates a smaller number of replicas for the majority of the microservices as compared to the

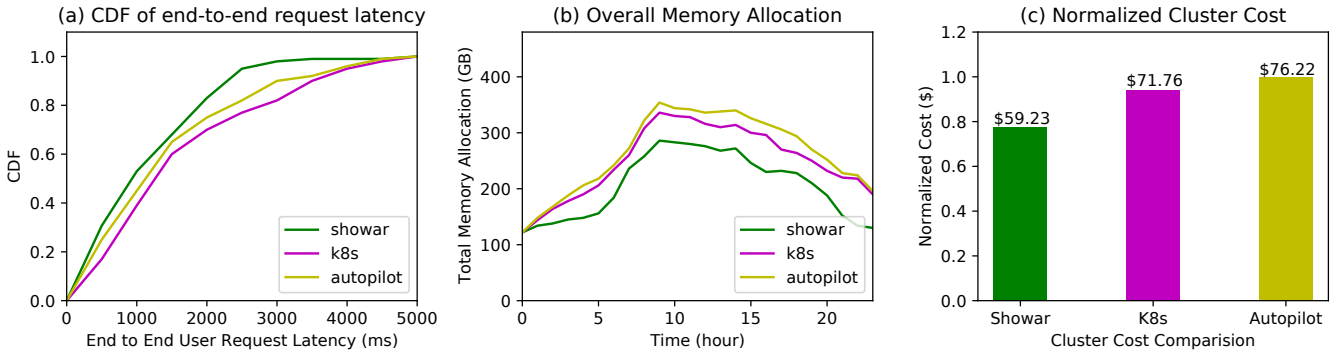

Figure 8. End-to-End Performance: (a) CDF of End-to-End Request Latency. (b) Total Cluster Memory Allocation. (c) Normalized Cluster Cost

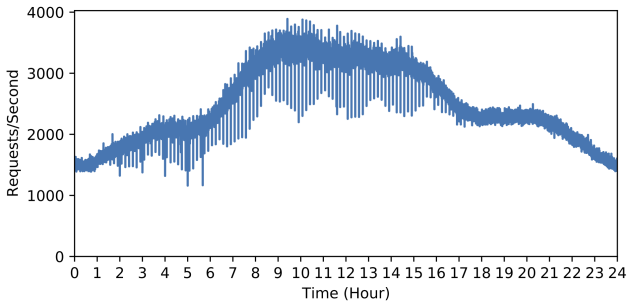

Figure 9. A 24-hours long workload from Wikipedia access trace.

baselines which are stateless, memoryless and reactive. More specifically, 95% of microservices have less than 9 replicas using *SHOWAR* compared to 15 replicas using Kubernetes and Autopilot. In addition, we see the effect of using more meaningful and representative metric in autoscaling decisions for individual microservices. In particular, during our evaluation, we observed that both Kubernetes and Autopilot typically set 16 replicas for *nginx* (a front-end microservice) primarily because of its high CPU utilization. However, as seen in Figure 2, a high CPU utilization does not always correspond to highly improved microservice performance. In contrast, *SHOWAR* sets only 10 replicas for this microservice. On the other hand, for the *User* microservice that several other microservices depend on it, both Kubernetes and Autopilot typically set only 3 replicas for it. In contrast, *SHOWAR* typically sets 6 replicas for this microservice.

5.4 The Effect of Affinity and Anti-Affinity Rules

Here, we evaluate the effect of Pod affinity and anti-affinity rules generated by *SHOWAR* using the correlation of CPU, memory, and network I/O usage between different microservices. We use the workload from Figure 4 and disable both vertical and horizontal autoscalers to see how our the generated affinity and anti-affinity rules can affect the Kubernetes' scheduler decisions on microservices and its effect on the latencies of users' requests compared to a situation where the scheduler does not use any affinity and anti-affinity rules. Due to space limits, we don't show the correlation data between the different microservices over the course of the experiments. Figure 7 depicts the CDF of the end to end user request's latencies. As it can be seen, by providing scheduling hints (using affinity and anti-affinity) for the scheduler, *SHOWAR* is able to improve the P99th latency that the users experience. In particular, using the affinity and anti-affinity rules generated by *SHOWAR*, the P99th of request latency is 6600 milliseconds compared to 9000 milliseconds using Kubernetes default scheduling decisions.

5.5 End-to-End performance

While we evaluated each component of *SHOWAR* individually in the past three subsections, here we enable all the three components to work in tandem and perform an end-to-end evaluation. We use a 24-hour long workload from the Wikipedia access trace shown in Figure 9 and hence each experiment lasts for 24 hours to capture all the patterns in the (real-world) workload. To fit the workload, we increased the size of our cluster to 30 VM instances. Our results show that *SHOWAR* improves the resource allocation and utilization while maintaining a comparable performance compared to the baselines.

Figure 8a depicts the CDF of the end to end request latency experienced by the user during the 24 hours of the experiment. As it can be seen, the end to end performance using *SHOWAR* is comparable to the baselines and using its affinity and anti-affinity rule generator as well as its dependency-aware horizontal autoscaling, *SHOWAR* is able to improve the P99th latency by more than 20% compared to the Autopilot and Kubernetes. Both Autopilot and Kubernetes show similar performance in P99th latency, however, because of allocating more memory for the replicas, Autopilot generally outperforms the Kubernetes at lower tails.

Figure 8b shows the total memory allocation (i.e. sum of memory limits set for the microservices replicas) in the cluster during the course of the experiments. Compared to the baselines, *SHOWAR* allocates less memory for the microservices replicas on average. In particular, on average, *SHOWAR* allocated 205 GB, while Autopilot and Kubernetes allocated 264 GB and 249 GB respectively. As it was seen in subsection 3.1 and subsection 3.2, it is mainly because *SHOWAR*'s vertical autoscaler achieves lower memory usage slack and also its horizontal autoscaler sets a lower number of replicas for the microservices. As such, the total memory allocation using *SHOWAR* is less than the baselines.

Finally, in Figure 8c we show the normalized cluster cost for each experiment. We normalize the average memory allocation to the memory size of one virtual machine in the cluster (i.e. 16 GB for $m5.xlarge$ instances) and multiply it by the cost of one virtual machine (i.e. $\$0.192/hour$) in 24 hours. This is because, usually the VM's price on public clouds is a linear function of memory size [17]. As it can be seen, compared to the Autopilot and Kubernetes, *SHOWAR* improves the total cluster cost by 22% and 17% respectively. The improvements come from the fact that *SHOWAR*'s vertical and horizontal autoscalers allocate less amount of compute resources at a comparable performance compared to the baselines.

6 Limitations and Future Work

Generally, we designed *SHOWAR* to be computationally light weight and adaptable, in contrast with "black box" approaches that use machine learning that need training and fail to cope with workload shift, e.g., [39, 55, 57].

Nevertheless, a major limitation of *SHOWAR* currently is that it is reactive to the resource usages of the microservices. As a result, a proper avenue to explore is to equip *SHOWAR* with near-term workload and resource usage prediction, e.g., [18]. Combined with it's current design, predicting the near future workload can improve the *SHOWAR* 's resource allocation and prevent performance degradation due to inadequate autoscaling actions.

Another limitation that current *SHOWAR*'s design has is that it only focuses on microservices autoscaling and assumes a fixed-sized cluster. It's important to address scenarios where the total amount of resources that the application autoscaler requests is more than the total available cluster resources. While cluster autoscaling is orthogonal to application autoscaling, they need to work together to achieve both overall efficiency in resource allocation and the application's performance requirements. As such, a communication and coordination between the two autoscalers is required to add more resources to the cluster. In future work, we plan to improve *SHOWAR*'s autoscalers to work with existing cluster autoscalers [12].

SHOWAR is designed as an autoscaler for Kubernetes. As such, it can be used for autoscaling the workloads that are supported by Kubernetes such as user-facing microservices and long-running batch jobs. However, while Kubernetes supports short-running jobs such as serverless functions [29], autoscalers such as *SHOWAR* may not be usable for this type of workload. One reason is that vertical autoscaling is not applicable because the size of containers for serverless functions are predefined. *SHOWAR*'s horizontal autoscaler may face additional complexity, e.g., keeping track of the number of "dormant" serverless functions (which can be warm started) and the time until each of them "expires" (and so would require a cold start delay). We leave exploring control-theoretic approaches for horizontal scaling of serverless functions to future work.

Finally, we plan to improve the *SHOWAR*'s affinity and anti-affinity rule generators. Currently we determine affinity pairwise between microservices using simple empirical resource-utilization correlation coefficients. We can in the future explore, for example, the impact on affinities of other statistics such as cross-correlations between different types of resources, and explore different types of scheduling mechanisms that can exploit such "raw" statistical information directly toward more efficient resource utilization [19].

7 Conclusion

In summary, we propose *SHOWAR* a framework consisting of a vertical autoscaler, a horizontal autoscaler and a scheduling affinity rule generator for the microservices. *SHOWAR*'s vertical autoscaler embraces the empirical variance in the historical resource usage to find the optimal size and reduce the resource usage slack (the difference between allocated resource and actual resource usage). For horizontal scaling *SHOWAR* uses ideas from control theory along with kernel level performance metrics (i.e. eBPF *runq latency*) to perform accurate horizontal scaling actions. In particular uses proportional–integral–derivative controller (PID controller) as a stateful controller to control the number of replicas for each microservices. The vertical and horizontal autoscalers in *SHOWAR* work in tandem to improve the resource utilization while maintaining a good performance. Additionally, once the size for each microservices is found, *SHOWAR* bridges the gap between optimal resource allocation and scheduling by generating affinity hints for the task scheduler to further improve performance. Our empirical experiments using a variety of microservice applications and real-world workloads show that, compared to state of the art autoscaling systems, on average *SHOWAR* improves the resource allocation by up to 22% (and hence saving in costs) while improving the 99th percentile end-to-end user request latency by 20%.

Acknowledgments

We thank Mohammad Shahrad for his valuable feedback on this work. We also thank the anonymous reviewers and our shepherd, Jana Giceva, for helping us improve the paper. This work was supported in part by NSF CCF grant 2028929 and NSF CNS grant 2122155.

References

[1] April, 01, 2021. Adopting microservices at Netflix. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[2] April, 02, 2021. Docker Swarm. https://docs.docker.com/engine/swarm/.

[3] April, 02, 2021. Kubernetes. https://kubernetes.io/.

[4] April, 05, 2021. Microsoft Microservices Architecture Guide. https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.

[5] April, 07, 2021. bpftrace. https://bpftrace.org/.

[6] April, 07, 2021. eBPF. https://ebpf.io/.

[7] April, 07, 2021. Facebook Katran. https://github.com/facebookincubator/katran.

[8] April, 07, 2021. The Cilium Project. https://cilium.io/.

[9] April, 07, 2021. The Falco Project. https://falco.org/.

[10] April, 10, 2021. Adapt or Die: A microservices story at Google, December. https://www.slideshare.net/apigee/adapt-or-die-a-microservices-story-at-google.

[11] April, 12, 2021. Google Cloud Services Outage Due to Insufficient Resource Quotas. https://status.cloud.google.com/incident/zall/20013.

[12] April, 12, 2021. Kubernetes Cluster Autoscaler. https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

[13] April, 14, 2021. Autoscaling in Amazon Web Sevices Cloud. https://aws.amazon.com/autoscaling/.

[14] April, 14, 2021. Autoscaling in Google Cloud Platform. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling.

[15] April, 14, 2021. Autoscaling in Microsoft Azure Cloud. https://azure.microsoft.com/en-us/features/autoscale/.

[16] April, 21, 2021. Microservices Architecture on Google App Engine. https://cloud.google.com/appengine/docs/standard/python/microservices-on-app-engine.

[17] April, 22, 2021. AWS EC2 On-demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[18] April, 22, 2021. Netflix's Predictive Auto Scaling Engine. https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270.

[19] April, 25, 2021. Scheduling Extensions in Kubernetes. https://github.com/akanso/extending-kube-scheduler.

[20] Feb. 16, 2021. GCP kubernetes best practices. https://cloud.google.com/blog/products/containers-kubernetes/kubernetes-best-practices-resource-requests-and-limits.

[21] Feb. 16, 2021. Kubernetes horizontal pod autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[22] Feb. 16, 2021. Kubernetes vertical pod autoscaler. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

[23] Feb. 18, 2021. CPU Utilization is Wrong. http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html.

[24] Feb. 28, 2021. eBP runq latency metric. http://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html.

[25] Feb. 28, 2021. Prometheus Monitoring Tool. https://prometheus.io/.

[26] March, 20, 2021. Locust workload generator. https://locust.io/.

[27] March, 22, 2021. Kubernetes Autoscalers. https://github.com/kubernetes/autoscaler.

[28] March, 28, 2021. Google Cloud Platform's Demo Microservice. https://github.com/GoogleCloudPlatform/microservices-demo.

[29] September, 15, 2021. Kubeless. https://kubeless.io/.

[30] September, 15, 2021. Kubernetes Controller Pattern. https://kubernetes.io/docs/concepts/architecture/controller/.

[31] September, 15, 2021. Linux Kernel Scheduler sysctl_sched_latency. https://elixir.bootlin.com/linux/v4.6/source/kernel/sched/fair.c#L50.

[32] September, 22, 2021. Pearson correlation coefficient. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[33] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. 2019. BurScale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*. 126–138.

[34] Luiz André Barroso and Urs Hölzle. 2009. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 4, 1 (2009), 1–108.

[35] Brendan Burns. 2018. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services.* " O'Reilly Media, Inc.".

[36] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* (2013).

[37] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical & Scalable ML-Driven Performance Debugging in Microservices. (2021).

[38] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[39] Yu Gan, Yanqi Zhang, Kelvin Hu, Yuan He, Meghna Pancholi, Dailun Cheng, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI).

[40] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1994–2004.

[41] Manuel Gotin, Felix Lösch, Robert Heinrich, and Ralf Reussner. 2018. Investigating performance metrics for scaling microservices in cloudiot-environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 157–167.

[42] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press.

[43] J.A. Hellerstein et al. 2004. *Feedback Control for Computing Systems.* IEEE Press / Wiley.

[44] P.K. Janert. Oct. 2013. *Feedback Control for Computer Systems.* O'Reilly.

[45] Philipp K Janert. 2013. *Feedback control for computer systems: introducing control theory to enterprise programmers.* " O'Reilly Media, Inc.".

[46] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 783–798.

[47] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[48] J. Levin and T. A. Benson. [n.d.]. ViperProbe: Rethinking Microservice Observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*.

[49] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International*

Symposium on Computer Architecture.

[50] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing* 12, 4 (2014), 559–592.

[51] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2884–2892.

[52] C. MacCarthaig. Sept. 1, 2019. PID Loops and the Art of Keeping Systems Stable. https://www.youtube.com/watch?v=3AxSwCC7I4s.

[53] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*.

[54] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. 2020. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media.

[55] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association.

[56] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*. 1–13.

[57] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes. April 2020. Autopilot: Workload autoscaling at Google. In *Proc. ACM EuroSys*.

[58] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity at scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

[59] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. 2009. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks* 53, 11 (2009), 1830–1845.

[60] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.

[61] Eberhard Wolff. 2016. *Microservices: flexible software architecture*. Addison-Wesley Professional.

[62] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices. (2021).

[63] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 149–161.

[64] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* (2018).