

# Tiny Autoscalers for Tiny Workloads: Dynamic CPU Allocation for Serverless Functions

Yuxuan Zhao  
LIACS, Leiden University  
y.zhao@liacs.leidenuniv.nl

Alexandru Uta  
LIACS, Leiden University  
a.uta@liacs.leidenuniv.nl

**Abstract**—In serverless computing, applications are executed under lightweight virtualization and isolation environments, such as containers or micro virtual machines. Typically, their memory allocation is set by the user before deployment. All other resources, such as CPU, are allocated by the provider statically and proportionally to memory allocations. This contributes to either under-utilization or throttling. The former significantly impacts the provider, while the latter impacts the client. To solve this problem and accommodate both clients and providers, a solution is dynamic CPU allocation achieved through autoscaling. Autoscaling has been investigated for long-running applications using history-based techniques and prediction. However, serverless applications are short-running workloads, where such techniques are not well suited.

In this paper we investigate tiny autoscalers and how dynamic CPU allocation techniques perform for short-running serverless workloads. We experiment with Kubernetes as the underlying platform, and implement using its vertical pod autoscaler several dynamic CPU rightsizing techniques. We compare these techniques using state-of-the-art serverless workloads. Our experiments show that dynamic CPU allocation for short-running serverless functions is feasible and can be achieved with lightweight algorithms that offer good performance.

## I. INTRODUCTION

Serverless computing is gaining significant traction, showing large growths in the past years. Many applications are now running atop serverless offerings, such as machine learning [1], [2], video processing [3], [4], or analytics [5]–[7]. Serverless applications are composed of many short-lived functions running on top of virtualization layers such as containers [8] or lightweight virtual machines [9]. Throughout the paper, we will refer to both these techniques as *serverless containers*. Typically, serverless functions run for at most 15 minutes and as any other workloads exhibit dynamic resource demands [10], [11]. However, functions are allocated static amounts of CPU time, usually proportional to their memory allocation [12]. Static CPU allocation in conjunction with dynamic workloads opens up interesting avenues for dynamic autoscaling of serverless containers. Although autoscaling has been investigated for long-running cloud computing applications, it has not been considered yet for dynamic CPU allocation in serverless environments. In this paper we take a step toward rightsizing the CPU allocation of serverless containers dynamically, during runtime, through tiny autoscalers.

In serverless computing, providers run up to thousands [9] of serverless containers on a single server to make use of the economy of scale to improve resource utilization. Most

of the functions running on these containers are short running and infrequently invoked [13]. However, providers do keep containers around after an invocation has finished to improve cold start latencies for future invocations [9], [14] or predict when functions will be invoked to pre-warm containers [13]. Resource allocators and schedulers take into account also the resources allocated to idle containers, even though they might not be used. Therefore, idle containers that are allocated a large portion of CPU might contribute to the overall under-utilization of a server.

While running thousands of functions on a server can lead to CPU resource utilization issues, memory utilization is better studied. Users typically select how much memory their functions should be allocated [12]. While running thousands of these in a single server sounds prohibitive, techniques such as REAP already exist for reducing serverless containers memory footprint [14]. On the other hand, in serverless environments, CPU allocations are static and performed non-transparently to the user, proportionally to the amount of memory allocated. It is thus difficult for a non-technical user to achieve a good CPU allocation for a given application. Although they are short-running, serverless functions exhibit dynamic and non-trivial resource usage, which makes it difficult [15] for their authors to estimate correctly the amount of resources to be requested from the cloud provider.

A solution to this problem is given by rightsizing and autoscaling algorithms. Cloud-based autoscaling has been extensively studied [16]–[20] and many performance-related studies have compared autoscaling algorithms [21]–[23]. However, these are based on long-running cloud applications, where historical information is abundantly available for making accurate predictions. In our case, with short running functions that are infrequently invoked, such approaches are not well suited. First, there is little to no historical data available (i.e., in the case of cold starts). It would also be prohibitively expensive to store fine-grained resource usage information for all the functions a provider serves. Second, quick reactions to dynamic resource fluctuations are needed. Third, some of these algorithms are computationally expensive. Running them for thousands of serverless containers at once might be prohibitive.

Therefore, in this paper we investigate more lightweight rightsizing and autoscaling mechanisms, such as simple-moving average (SMA) and exponential moving average (EMA), inspired from web-based autoscaling techniques [24].

We call these tiny autoscalers and compare them with recent methods on container autoscaling [25] based on Holt-Winters exponential smoothing [26] and long short-term memory (LSTM) [27]. We show the implications of tiny autoscalers and how these can be leveraged by practitioners.

Without loss of generality, we implement tiny autoscalers on top of Kubernetes [28], a container orchestration engine introduced by Google. Similar techniques could be implemented for lightweight virtual machines [9] as well using mechanisms such as *Linux cgroups*. Kubernetes makes use of an autoscaling recommender which we override to implement tiny autoscalers. We further compare these with the default Kubernetes autoscaler which was designed for the Google Borg Autopilot [29]. Our experiments show that the default autoscaler is not well suited for short-running serverless workloads and that significant over- and under-utilization is exhibited through the default Kubernetes autoscaler when running serverless workloads.

Toward showing that dynamic CPU allocation for serverless functions is achievable, our contributions are the following:

- 1) We design, implement, tune and release as open-source tiny autoscalers: lightweight autoscaling mechanisms for serverless workloads. We showcase existing issues in existing state-of-the-art autoscalers that act as barriers in applying them for serverless workloads. (Section III).
- 2) We show empirically that dynamically allocating CPU for short-lived serverless functions through tiny autoscalers is feasible and efficient (Section IV).
- 3) We discuss the implications of our results for the design and feasibility of tiny autoscalers for dynamically allocating CPU for serverless workloads (Section V).

## II. SYSTEM MODEL

We make use of the Kubernetes engine to run serverless containers and applications and dynamically alter their allocated CPU during runtime. We describe how Kubernetes works and focus on its default container autoscaler, the Vertical Pod Autoscaler (VPA) Recommender.

The basic architecture of our system is shown in Figure 1. Our experiment is conducted on a minikube cluster on Ubuntu 20.04. On this minikube cluster, pods are deployed and we configure only one container running per pod. The pods are monitored to record their resource usage in a per-second granularity. The data is collected in a MongoDB database. Furthermore, we enable VPA to provide a mechanism for dynamically resizing the containers resource request. The main aims of VPA are not only reducing the redundant resource waste requested by containers but also reducing the probability of an application in the container being throttled or terminated due to insufficient resources. The VPA primarily consists of three components: recommender, updater, and admission controller. In this article, we mainly focus on the recommender component in VPA. The autoscaling algorithms are integrated into the recommender component and the performance of our algorithms is validated by configuring the VPA with custom autoscaling algorithms. In the following sections, we

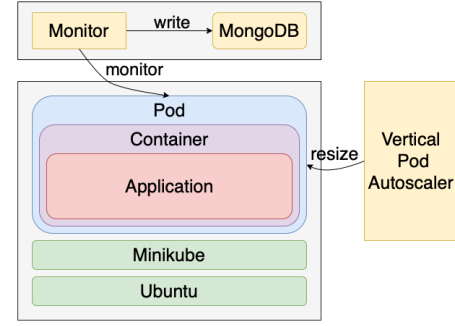


Fig. 1. The architecture overview of our system. We deploy a pod containing the application on minikube. The pod is monitored and the monitoring data is stored in a MongoDB database. A VPA is attached to the pod for resizing.

TABLE I  
DATA MONITORING SOURCE.

Data	API
Resource usage	client.CustomObjectsApi
Resource requests	client.CoreV1Api
VPA recommendations	client.ApiClient

demonstrate in detail how we monitor the resource usage in Kubernetes and show an overview of VPA architecture [30].

### A. Resource Monitoring

The resource metrics pipeline [31] of Kubernetes reports values through the underlying *Linux cgroups*. These monitoring data are collected by cAdvisor (container advisor) [32], which is a project open-sourced by Google. cAdvisor can collect the information of all the running containers on a machine, including CPU usage, memory usage, etc. Then, cAdvisor is integrated into kubelet [33]. Kubelet is an agent running on each node in the cluster. Thus, the metrics server [34] gets the resource metrics from kubelet and integrates the resource metrics into an API-server, such as the metrics API [35].

CPU and memory usage of containers are monitored through the *kubernetes.client.CustomObjectsApi* in the Kubernetes python client. Resource requests of pods and containers are read via *kubernetes.client.CoreV1Api*. Resources recommended by VPA are obtained from *kubernetes.client.ApiClient*. Table I lists three data types and their corresponding APIs where we monitor the resource values. All of the resource values mentioned above are monitored every second with fine granularity and stored in a MongoDB database in real-time. The CPU resource is measured in milliCPU (or millicores), a unit of measure introduced by Kubernetes. For example, 100m CPU is equivalent to 0.1 CPU.

### B. Vertical Pod Autoscaler

The VPA recommender is an essential component in the VPA. It provides the core resource usages (CPU and memory) estimation algorithm which recommends appropriate resource request values for pods in Kubernetes. As a result, the containers would resize according to the recommended resource values. Therefore, the quality of the recommendation algorithm largely determines the quality of the container resizing. An application would be throttled if it exceeds the specified CPU limit of the container and gets terminated if its memory exceeds the limited amount of the container.

TABLE II  
METHODS CONSIDERED AND COMPARED IN THIS ARTICLE.

Methods	Integrated with Kubernetes VPA
VPA recommender	Yes (existing)
HW recommender	No, emulation
LSTM recommender	No, emulation
SMA recommender	Yes (our contribution)
EMA recommender	Yes (our contribution)

The VPA is mainly composed of recommender, updater, and admission controller. The recommender watches all the pods in the Kubernetes cluster and calculates the recommendation values for each pod. The recommendation algorithm is included in section III-A. The recommender reads the pod metrics from Prometheus [36] regularly. The updater is responsible for updating pods according to pod recommendations. Currently, the VPA updater only supports evicting old pods before recreating a new pod with the recommended resources. This means the service will be disrupted when a pod needs an update. In-place update [37] was proposed, but it is still under development. In this paper we turn off the updating mechanism and only consider the recommendations, thus not disrupting containers. This is sufficient to showcase the contributions of this article.

### III. AUTOSCALING METHODS

We discuss autoscaling methods and explain how one can adapt them to the problem of dynamically rightsizing the CPU allocation of serverless functions. We identify drawbacks for these techniques and show how we overcome these.

We first introduce the resource estimation algorithm in the original Kubernetes VPA. The recommendation algorithm currently used in vertical pod autoscaler is deeply inspired by the moving window recommender in Google Borg Autopilot [29]. Then we discuss the autoscaling strategy proposed in [25], which primarily applies Holt-Winters exponential smoothing (HW) [26] and Long Short-Term Memory (LSTM) [27] algorithms to predict the future resource demands. Lastly, we present a resource estimation algorithm largely based on the ideas in CPU usage prediction models in web-based system [24] and bring forth new ideas of how to adapt these algorithms to the problem at hand. We improve the algorithms to adapt to the scenario of serverless function CPU resizing.

Table II gives a summary of the methods used in this paper. Besides our newly implemented algorithms, the original VPA recommendation algorithm is already implemented into vertical pod autoscaler. As for HW and LSTM autoscaling strategy proposed in [25], they are not integrated into the vertical pod autoscaler component by their authors and it is outside the scope of this paper to implement them in the Kubernetes VPA. We instead use emulation to compare these algorithms with tiny autoscalers.

#### A. Kubernetes VPA Recommender

The recommender of the VPA mainly borrows the ideas of the moving window recommender in Google Borg Autopilot [29]. The VPA recommender creates a decaying histogram object to store the CPU usage for every container.

The recommender acquires the resource usage of all pods from Prometheus [36] regularly and writes the resource usage of containers into a maintained corresponding decaying histogram. The decaying histogram is composed of multiple buckets, which are used to store the weight of resource usage. The size of buckets in a decaying histogram grows exponentially with a ratio of 1.05. The first bucket stores the weights of resource usage in the range of  $[0, firstBucketSize)$ . Since the bucket size grows exponentially, the size of the  $n$ th bucket follows  $firstBucketSize * ratio^{n-1}$ . The weight of every resource usage is stored in the bucket where the resource usage falls between the bucket boundaries. As time goes by, usage weight decreases as well. If the default *HalfLife* is set to 24h, the weight of the past 24 hours before will be halved.

In terms of recommendation, the VPA recommender uses three values: target value, lower bound value, and upper bound value. They are the starting values (left boundaries) of the bucket where the total weight of that bucket and the former buckets arrives at  $0.9 * totalWeight$ ,  $0.5 * totalWeight$ , and  $0.95 * totalWeight$  for the first time, correspondingly. Furthermore, VPA recommender applies a confidence multiplier to lower bound value and upper bound value. The VPA evicts the pod as soon as its request value is beyond the range of upper bound and lower bound, then creates a pod with the current recommended value as its request.

**Kubernetes VPA recommender drawback:** The Kubernetes default VPA recommender, based on Google Borg Autopilot [29], is not sensitive to short, sudden changes in workloads, and is more suited for longer running workloads. Serverless workloads are short running with sudden bursts in CPU usage.

#### B. HW and LSTM Recommender

Wang et al. proposed an autoscaling mechanism [25] which applies Holt-Winters exponential smoothing (HW) [26] and Long Short-Term Memory (LSTM) [27] algorithms to increase the CPU utilization of Kubernetes containers. Their autoscaler takes target value, lower bound value, and upper bound value from HW and LSTM models as input and supplies 120 millicores as the error buffer. Their algorithm will give a new recommended value when the current CPU request is out of the range of bounds. They preset two values to avoid unnecessary rescaling. One is the rescaling cool-down value, the other is the minimum change check value. This means it will rescale only after a cool-down time has passed since the last rescaling. In addition, the difference between the value of the current request and a new request must be more than minimum change check value. HW and LSTM recommenders are implemented in Python with *Statsmodels* and *Keras*. For HW recommender, the prediction model refits when each new observation is collected. For LSTM recommender, the prediction model retrains every season (several observations compose a season). At least two season data are needed to generate the future CPU usage prediction from HW and LSTM recommenders because these two models need to be initialized at the beginning. That explains why there is a fixed

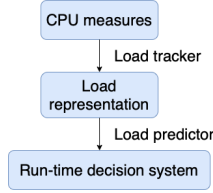


Fig. 2. The basic framework of load prediction models [24]. The actual CPU usage are processed by two types of load trackers and get CPU usage representations, then make prediction for future CPU demands according to the representations.

recommendation value in cold starts for both HW and LSTM recommenders (See Section IV-D).

**Emulation.** Unfortunately, this autoscaling mechanism is not integrated into the VPA. Thus in this article the performance of HW and LSTM is not benchmarked in real-time but rather in emulation. We initially run the experiments without any VPA and collect the CPU usage data. Then we train the HW and LSTM models using a subset of the initial data. The season length is set to one minute and as training data we use two seasons. Using any shorter seasons will result in very poor performance. The input data are fed into these two models to refit or retrain the models and get the predicted values of future CPU usage. The final output of these models is the recommended value for the future CPU demands.

The major downside of the HW and LSTM recommenders is that they need significant amount of training data to perform well. In short running serverless workloads, this is either difficult to get since the functions are very short, or very expensive to store since providers are running extremely large numbers of different functions. Keeping utilization data for all of them will be prohibitive. Hence, The HW and LSTM recommenders are expensive for serverless containers.

**LSTM and HW drawback:** ML-based recommenders need significant amount of data for training, which might not be available in serverless scenarios. Moreover, ML-based recommenders are computationally expensive.

### C. Tiny Autoscalers: SMA-and EMA-based

Andreolini et al. [24] proposed load prediction models for CPU usage in web-based systems. Since these models are lightweight to apply in terms of computational complexity and do not need much training data, in this paper we investigate if they are suitable for short-lived serverless workloads as tiny autoscalers. We innovate these algorithms while inheriting the ideas behind their prediction models. We adapt the models and implement them in the Kubernetes VPA.

Andreolini et al. demonstrate in their paper [24] that it is infeasible to predict future load well using the raw CPU usage measures. They design load trackers, two linear functions to smooth the trend of CPU usage, representing the CPU load behavior of the system (see Section III-C1). Then, they make predictions of CPU usage through load representations processed by load trackers. The basic framework of the load prediction models is depicted in Figure 2.

1) *Load trackers:* As Figure 2 shows, the load prediction model has a two-step approach. In the first phase of the models, we still apply two linear load tracker functions presented in the original paper [24], which are simple moving average (SMA) load tracker and exponential moving average (EMA) load tracker. Given a CPU usage value  $s_i$  measured at time  $t_i$  and previously sampled  $n$  CPU usage values, they compose a set  $S_n(t_i) = (s_{i-n}, \dots, s_i)$ . The load tracker function is defined as  $LT(S_n(t_i)) : \mathbb{R}^{n+1} \rightarrow \mathbb{R}$ , where the  $S_n(t_i)$  is the input of the load tracker function and it returns a representation  $l_i$  to represent the set of  $S_n(t_i)$  at time  $t_i$ . Simple moving average (SMA) is the unweighted average of  $n + 1$  CPU usage values in the set  $S_n(t_i)$ , the weights assigned to each observation are the same. So the SMA-based load tracker function at time  $t_i$  is defined as:

$$SMA(S_n(t_i)) = \frac{\sum_{i-n \leq j \leq i} s_j}{n+1}$$
 The problem of the SMA load tracker in theory is that it will involve a delay when it represents the workload trend, especially if the size of  $S_n(t_i)$  is large. As opposed to SMA, exponential moving average (EMA) load tracker function can decrease the delay effect well in theory. Exponential moving average (EMA) is the weighted average of  $n + 1$  CPU usage values in the set  $S_n(t_i)$  and the weights of observations are exponentially decreasing. Thus, the EMA-based load tracker function at time  $t_i$  is defined as:

$$EMA(S_n(t_i)) = \begin{cases} \frac{\sum_{0 \leq j \leq n} s_j}{n+1} & \text{if } i \leq n, \\ \alpha * s_i + (1 - \alpha) * EMA(S_n(t_{i-1})) & \text{if } i > n. \end{cases}$$

The parameter  $\alpha$  is called the smoothing constant. We conform with the constant value in the paper [24] and set the smoothing constant  $\alpha = \frac{2}{n+1}$ . For the load tracker based on EMA at time  $t_i$ , the recent observations contribute more to the representation  $l_i$  than the older observations due to the decaying weights.

2) *Load predictor:* In the second phase of the models, we adapt and innovate the load prediction in the paper [24] according to our usage scenarios. The load predictor function is defined as  $LP(L_q(t_i)) : \mathbb{R}^{n+1} \rightarrow \mathbb{R}$ , where  $L_q(t_i) = (l_{i-q}, \dots, l_i)$  is a set of  $q + 1$  representations obtained from load tracker function.  $LP(L_q(t_i))$  returns the predicted future CPU usage value.  $LP(L_q(t_i)) = \text{Max}(\beta * S/EMA(S_n(t_i)), m * (i + k) + a)$ , where  $m = \frac{l_i - l_{i-q}}{q}$ , and  $a = l_{i-q} - m * (i - q)$ .

**EMA and SMA drawback:** While EMA and SMA algorithms are lightweight and effective in following the resource demand, we experimentally found that EMA and SMA need to be tuned for the workloads they target. As a consequence, we adapt EMA and SMA to work efficiently on serverless workloads. In Section IV-C we show how our tuned versions perform.

The fundamental idea behind our tuning and modification for the load predictor is an heuristic which will always try to slightly over-provision than under-provision, so that the client

TABLE III  
SERVERLESS WORKLOADS USED IN OUR EXPERIMENTS.

Name	Input Size	Runtime
image_rotate	10,000 Images	4 minutes
image_rotate_shorter	1,000 Images	15 seconds
lr_training_default	5,700MB CSV	10 minutes
video_processing_17m	17MB Video	15 seconds
video_processing_67m	67MB Video	1.5 minutes
video_processing_127m	127MB Video	4 minutes

TABLE IV  
METRICS INTERVAL INFLUENCE ON DEFAULT VPA OVER-PROVISIONING.

Metrics Interval	Average Slack (millicores)
1 minute	426.23
1 second	328.23

serverless function will not be significantly throttled. Thus, in our prediction algorithm, we introduced  $\beta$ , a constant to multiply with the former average of recent CPU usage values. We then take the maximum between this term and the original predicted value. This mechanism is a *bottoming* mechanism so that the prediction does not drop very rapidly.

Since in the original formula we have the term  $m = \frac{l_i - l_{i-q}}{q}$  in the prediction,  $m$  will be close to 0 when the CPU load tends to be flat, which will lead to a *cliff* on the predicted values. Thus, we introduce a novel *bottoming* mechanism to the EMA and SMA algorithms. The adoption of the bottoming mechanism prevents the predicted resource usage value from rapid and unexpected decline. Additionally, the linear extrapolation prediction, according to recent representations from load tracker, ensures the predicted value to rise abruptly when CPU usage is at peak. SMA-based and EMA-based recommenders have the advantage of their low computational complexity, so they are suitable for tiny autoscalers for serverless functions while keeping a promising prediction result.

In terms of update policy, we follow the design in the VPA keeping the lower bound and upper bound for a recommendation value. However, what is different from VPA is that we revise the calculation methods of lower bound and upper bound. In VPA, the lower bound and upper bound are calculated as the starting values of the bucket where its accumulated weight achieves 50% and 95% of total weight. In SMA- and EMA-based recommender, the upper bound and lower bound follow the trend of predicted value and with a multiplier as in the VPA. Thus, the lower bound and upper bound will converge to the predicted value as time goes by in the same as the default VPA. When the request value of the pod is out of the range of lower bound and upper bound, the pod will be updated with a new request value that is the same as the recommended value at that time.

Two parameters are needed in both SMA-based and EMA-based recommenders, which are the size of the load tracker and the number of the load trackers. The size of the load tracker represents how many observations are in a load tracker to calculate the (un)weighted average. The number of load trackers indicates how many representations  $l_i$  obtained from the load tracker function are used to make an extrapolation prediction. For example, we use EMA5-3 to represent EMA method with load tracker of size 5 and with 3 load trackers.

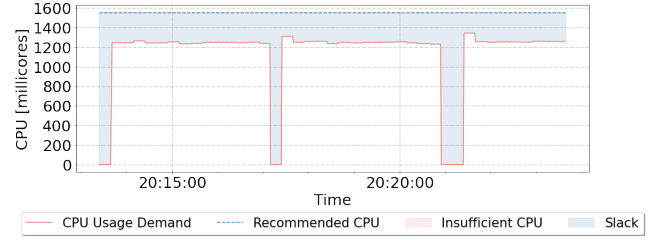


Fig. 3. Result of video\_processing\_127m with default VPA (metrics interval 1 minute). On this workload, the default VPA recommendation stays unchanged and cannot react with changes of the actual CPU usage.

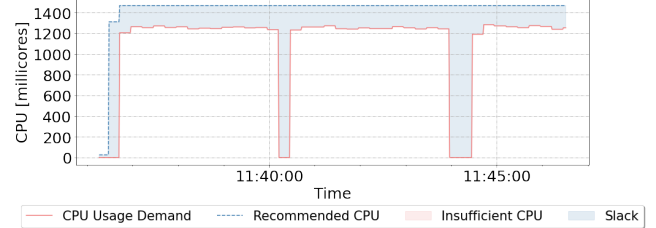


Fig. 4. Result of video\_processing\_127m with default VPA (metrics interval 1 second). The default VPA with metrics interval as 1 second can follow the trend of the actual CPU usage at the beginning but stays unchanged in the following warm starts.

#### IV. EXPERIMENTAL EVALUATION

We investigate empirically what are implications of dynamically autoscaling the CPU allocations of serverless containers. We explain our experimental setup, tune the default Kubernetes VPA, show our results tuning EMA- and SMA-based autoscalers. Finally, we show what are the implications of dynamically rightsizing CPU allocations using 4 different autoscalers for serverless workloads in both cold and warm starts. We discuss the practical implications of our results.

##### A. Experiment Setup

In this paper, we investigate the performance of several CPU autoscaling mechanisms on serverless workloads. These CPU autoscaling mechanisms include the default VPA in Kubernetes, simple moving average (SMA) and exponential moving average (EMA) which are inspired from web-based load prediction models [24], and container autoscaling methods [25] based on Holt-Winters exponential smoothing (HW) and long short-term memory (LSTM). The performance of these mechanisms are measured by two metrics: average slack and average insufficient CPU. Slack is the amount of CPU usage that is over-provisioned, while insufficient CPU usage refers to the amount of CPU usage that is under-provisioned. The metrics are defined in detail by Wang et al. [25] and can be viewed as a simplification of the extensive set of metrics defined by Ilyushkin et al. [21].

To evaluate these autoscalers we use three workloads from the vHive [14] archive. By using different input sizes which shorten or lengthen their running time we reach a total of six serverless workloads. According to the Microsoft investigation of their own serverless workloads [13], our proposed workloads are realistic in terms of runtime. Table III shows the serverless workloads evaluated in this paper and their runtime.



TABLE V  
COMPARING AVERAGE SLACK AND AVERAGE INSUFFICIENT CPU OF  
EMA5-3 BEFORE AND AFTER OUR TUNING AND ADAPTATION.

Tuning	Avg. Slack (millicores)	Avg. Insuf. (millicores)
Before tuning	5.16	288.75
After tuning	134.81	66.89

We run all our experiments under Ubuntu 20.04 and minikube v1.24.0. Our servers are an on-prem machine with 8 cores and 16GB RAM and *i3.metal* AWS EC2 virtual machine, which has 72 cores and 512GB RAM.

### B. Tuning the Kubernetes Default VPA Autoscaler

The default Kubernetes VPA Autoscaler makes recommendations at fixed-time intervals. The default interval is 1 minute. This is insufficient for serverless workloads which could be much shorter than 1 minute as shown by Shahrad et al. [13]. To improve the default Kubernetes VPA autoscaler we have tweaked this value to 1 second. We show the difference between the two time intervals in the following experiment.

To evaluate the performance of the default VPA autoscaler, we run all the workloads repeatedly. The first run of a workload emulates a cold start, while the subsequent runs emulate warm starts of the serverless functions. Figures 3 and 4 plot our results for the video processing workload. Interpreting these results, one could notice that the VPA with 1 second interval acts faster at the beginning because the first recommendation from autoscaler with 1 minute metrics interval only can be caught after 1 minute. Although both of them cannot follow the trend of CPU usage fluctuations, the autoscaler with 1 second metrics interval has less slack than autoscaler with 1 minute metrics interval. Table IV shows the average slack of default VPA autoscaler with different metrics intervals.

**Conclusion-1:** The default Kubernetes VPA cannot follow the actual CPU usage fluctuations closely. With 1 second metrics interval it exhibits less slack than for 1 minute metrics interval. Short-running serverless workloads need different autoscalers (Figures 3, 4, Table IV).

### C. Building Tiny Autoscalers: Tuning EMA and SMA

In Section III we introduced the SMA and EMA methods for web-based workload CPU usage prediction [24]. We have found that in their original form they do not perform well for serverless workloads, exhibiting significant under-provisioning. Thus, in this paper, we adapt these methods to serverless workloads as explained in Section III. To show the under-provisioning behavior more clearly, in this section we use a different workload, namely the YSCB [38] key-value store benchmark running on top of the Redis key-value store. The VPA we implemented monitors the Redis Kubernetes Pods and adapts their CPU dynamically.

As Figure 5 shows, the prediction of the unaltered original method can react with the change of CPU usage trend very fast, not only for increasing CPU but also for decreasing CPU. However, as the figure shows, this encompasses significant amounts of insufficient CPU (i.e., under-provisioning). This

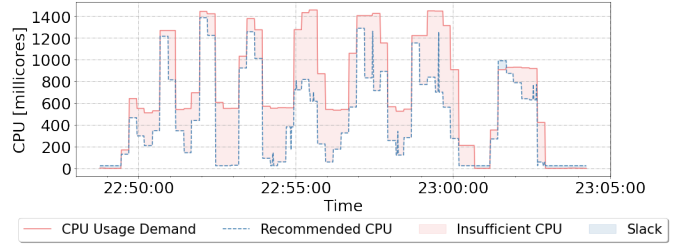


Fig. 5. Default ema5-3 dynamic CPU allocation (before tuning). Notice how the default EMA method consistently allocates insufficient CPU.

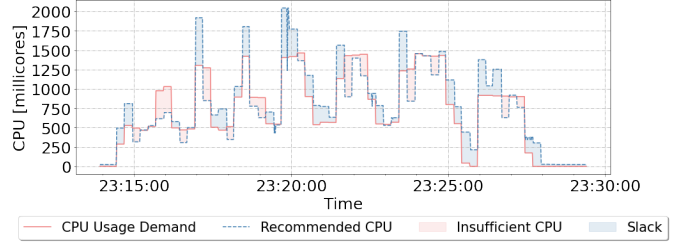


Fig. 6. Dynamic CPU allocation using ema5-3 including our tuning defined in Section III. Notice how the tuned EMA version has a much more balanced CPU allocation.

results in the application being slowed down. To overcome this disadvantage, we introduce a *bottoming* mechanism to the algorithms. We involve unweighted or weighted average of recent CPU usage values to prevent the prediction from rapid decline unexpectedly and unreasonably. As Figure 6 indicates, after our adaptation and tuning, the prediction is improved when the actual CPU usage decreases suddenly. Table V compares average slack and average insufficient CPU of ema5-3 before and after our adaptation.

In this section we present only the results of the ema5-3 method due to space limitations, but the results we have are consistent over all EMA and SMA methods, e.g., ema10-5, ema3-2, sma3-2, sma5-3. In the latter experiments presented in this section we show overall results using all these methods which perform better than the default Kubernetes VPA and the ML-based methods of LSTM and HW.

**Conclusion-2:** The tiny autoscalers can predict the actual CPU usage closely after our tuning. Our adaptation is able to leverage much less under-provisioning, offering the application better overall CPU performance (Figures 5, 6, Table V).

### D. Dynamic CPU Allocation for Cold Starts

One very important area of research related to serverless systems are cold starts. They refer to the first start of a serverless function on a given server, when the underlying subsystem is not pre-warmed (e.g., microVMs are not yet booted, runtimes are not loaded etc.). The first start might refer to either the first invocation of a function after its creation, or the first start of a function on a specific server. The behavior would be similar in both cases.

For scheduling containers in serverless workloads, cold starts are important because schedulers that make use of historic information, such as the default Kubernetes VPA, LSTM or HW discussed earlier in this paper, do not have any historical information. Even though a function might have

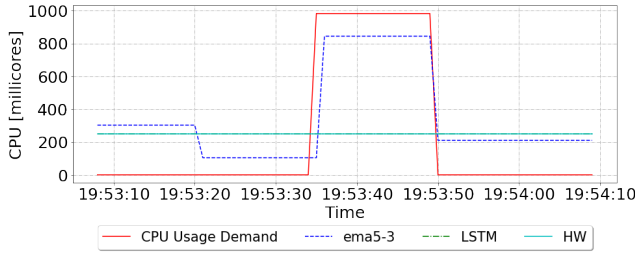


Fig. 7. Cold starts for HW and LSTM compared to ema5-3 when running image\_rotate\_shorter workload. Due to lack of historical information and training, LSTM and HW cannot allocate sufficient CPU. LSTM and HW curves overlap.

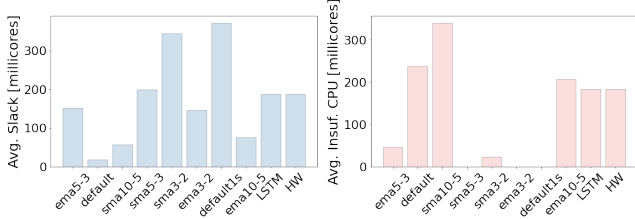


Fig. 8. Comparing the average slack and insufficient CPU among all the methods in this paper when running image\_rotate\_shorter workload in the cold start. Due to the training process of LSTM and HW, they do not perform well in both slack and insufficient CPU as a result of presenting a preset value.

been run previously on a different server, we assume that for cloud providers keeping fine-grained scheduling and resource usage information is prohibitive, as providers are likely to run millions of such functions every hour [13]. Under such auspices, it is important to know how autoscalers fair for the first time when scheduling a specific function.

We experimented with all applications and all autoscalers dynamically allocating CPU for cold application starts. We apply ML-based autoscalers, LSTM and HW using the same data on CPU usage as in the ema5-3 run for a more equitable performance comparison between EMA-based autoscaler and them. We present the behavior under cold starts for the image\_rotate\_shorter and video\_processing\_17m workloads in Figures 7 and 9. These figures show that basically the ML-based autoscalers, LSTM and HW, cannot allocate CPU dynamically for cold starts. This is because they do not have any historic information on which the methods could have been trained. Instead, they simply offer a default static amount of CPU. The EMA method we tuned in our previous experiment is able to match the CPU demand accurately.

For all the autoscalers we have implemented for this paper, we show their average slack and insufficient CPU for the cold starts of the two workloads, image\_rotate\_shorter and video\_processing\_17m, in Figures 8 and 10 correspondingly. ML-based autoscalers present high average insufficient CPU on two workloads (especially in Figure 10). It is immediately clear that the ML-based autoscalers cannot follow the CPU demand of the workloads during cold starts due to insufficient historical information. Similarly for the default Kubernetes VPA autoscaler, either significantly under-provisioning or over-provisioning for certain applications, presenting high average slack or average insufficient CPU in Figures 8 and 10. Tiny autoscalers achieve extremely low average insufficient CPU but also relatively low average slack at the same time.

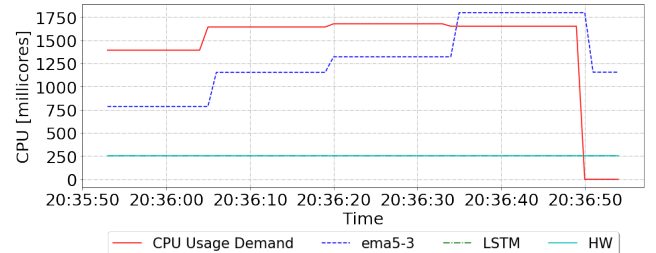


Fig. 9. Cold starts for HW and LSTM compared to ema5-3 when running video\_processing\_17m workload in the first run. Due to lack of historical information and training, LSTM and HW cannot allocate sufficient CPU. LSTM and HW curves overlap.

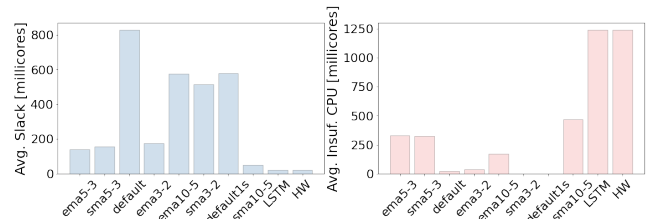


Fig. 10. Comparing the average slack and insufficient CPU among all the methods in this paper when running video\_processing\_17m workload in the cold start. Due to the training process of LSTM and HW, they do not perform well in both slack and insufficient CPU as a result of presenting a preset value.

**Conclusion-3:** For cold function starts, ML-based autoscaling algorithms achieve poor performance as they do not have sufficient data to be trained with. The tiny autoscalers can follow the CPU resource demand curves more closely, offering better performance (Figures 7, 8, 9, 10).

#### E. Dynamic CPU Allocation for Warm Starts

Microsoft shows that in their serverless system [13], many functions are invoked several times every hour and significant numbers of functions are actually invoked every few minutes. In these conditions, autoscalers that keep track of history have sufficient data to perform well in subsequent functions runs, or warm starts. In this section we investigate how the autoscalers perform when functions are invoked repeatedly, and whether historical information can help in taking better resource allocation decisions.

We run all the workloads under all autoscalers repeatedly for a period of tens of minutes. The first run emulates a cold start, while subsequent runs emulate further warm starts. Using this method we evaluate how the investigated autoscalers perform under warm starts and how accurately they can dynamically allocate CPU to applications.

Figure 11 shows the curves of the performance on the video processing workload for the EMA-based method, HW, and LSTM methods. From Figure 11, the curves indicate that the EMA-based method can follow the trend of CPU usage very well as opposed to methods based on HW and LSTM. Unlike HW and LSTM methods, the EMA-based method does not need data to train the model at the beginning.

We summarize the performance of all the the autoscalers over all applications for warm starts in Figures 12 and 13. The former presents the average slack (i.e., over-provisioning) and the latter presents the average insufficient CPU (i.e.,

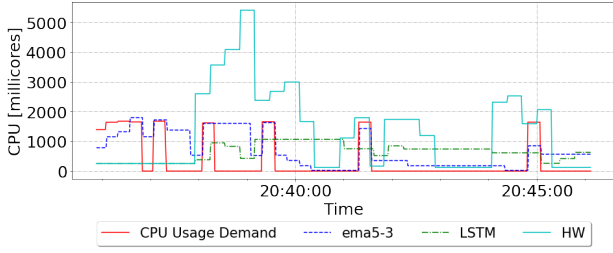


Fig. 11. Comparing HW and LSTM with our ema5-3 running on video\_processing\_17m workload. Notice our method follows the CPU usage trend more closely than HW and LSTM, offering better performance.

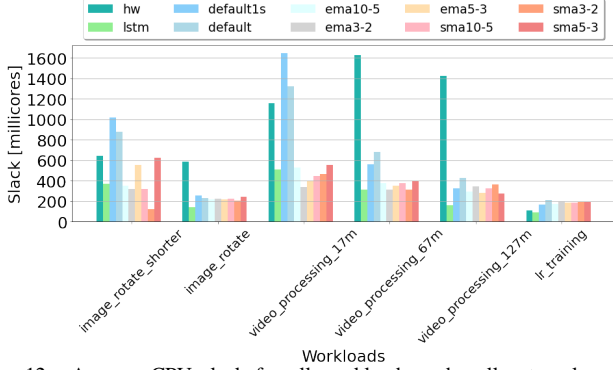


Fig. 12. Average CPU slack for all workloads under all autoscalers. The applications were run several times to emulate serverless warm starts.

under-provisioning). For all applications, the best performing autoscalers are the SMA and EMA-based ones that we have tuned. While the default Kubernetes VPA and LSTM and HW autoscalers present extreme behavior—in some situation very high slack and very low insufficient CPU—the EMA and SMA based autoscalers are able to keep both metrics relatively low. This is important because it shows conservative behavior that can help cloud practitioners offer good performance to clients without greatly over-provisioning. Conversely, the amount of application throttles is also kept to a minimum.

**Conclusion-4:** For warm function starts, tiny autoscalers offer better performance than ML-based autoscaling, following the CPU resource usage more closely and offering less average slack and insufficient CPU usage (Figures 11, 12, 13).

#### F. Running Thousands of Tiny Autoscalers

In practice, to achieve dynamic CPU allocation in serverless clouds, providers have to attach an autoscaler to every running function instance. Recent literature [9] shows that providers can run up to thousands of functions per server. In this experiment we show what the added CPU utilization overhead is for running thousands of tiny autoscalers per machine, one for each function.

To only measure the CPU overhead of the autoscaling mechanisms, we create Kubernetes containers that simply sleep after being booted. We then attach to each of these containers a VPA running the EMA5-3 autoscaler. Since the containers are only sleeping, the CPU utilization is only caused by the autoscalers running next to the containers.

Officially, Kubernetes developers have put a limit of 500 VPAs running on a single node. Unfortunately, we could

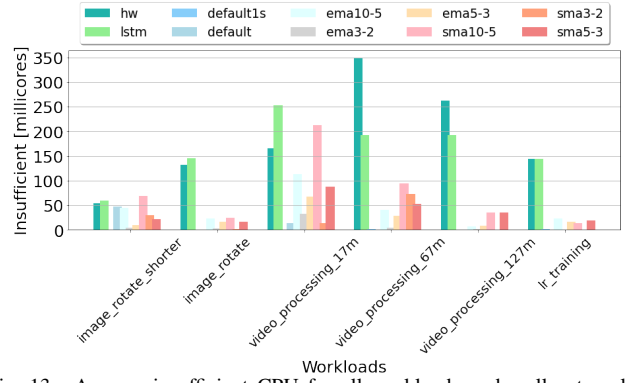


Fig. 13. Average insufficient CPU for all workloads under all autoscalers. The applications were run several times to emulate serverless warm starts.

not reach this limit and we could only run reliably up to 250 VPAs. This is not a serverless problem per se, but rather a Kubernetes limitation which does not affect the entire field. In future releases of Kubernetes this issue will be fixed. For this paper, going beyond 250 VPAs, we have used a polynomial curve fitting technique. Our results are plotted in Figure 14. Following the CPU utilization curve, 250 VPAs do not use more than 5% CPU. Our extrapolation shows that going up to 2,000 VPAs will not result in more than 17.5% CPU usage. This leaves sufficient room for the applications to run. If providers consider this limit too high, there are several options to reduce the load: (i) wrapping autoscalers in cgroups with CPU limits; (ii) use a coarser grained monitoring interval, or (iii) group multiple containers (e.g., containers from the same user) under a single autoscaler.

**Conclusion-5:** It is feasible to run thousands of tiny autoscalers, as these fine-grained and lightweight methods do not use much CPU (Figure 14).

#### V. SUMMARY: IS DYNAMIC CPU ALLOCATION FEASIBLE FOR SERVERLESS THROUGH TINY AUTOSCALERS?

Based on the results we show in Section IV, we return to our main question: is dynamic CPU allocation feasible in serverless environments? This is opposed to common current practice where CPU allocation is fixed and proportional to the memory allocation requested by the client.

##### 1. Are state-of-the-art autoscalers enough for serverless?

Currently, the default Kubernetes VPA can not solve the problem of recommendations for sudden and short-lived CPU usage increases very well. The recommendations from the default VPA usually have significant slack, which results in resource under-utilization. The default VPA was designed with different goals, such as longer-running workloads, which it is able to serve well as shown by previous work. Moreover, the ML-based autoscalers, such as LSTM and HW need significant amounts of training data to perform well. Such data might not be available for serverless workloads or might be prohibitively expensive to store at such a large scale.

##### 2. What is a good tiny autoscaler for serverless?

For this work we modified the original SMA- and EMA-based autoscalers to slightly over-provision, but within certain



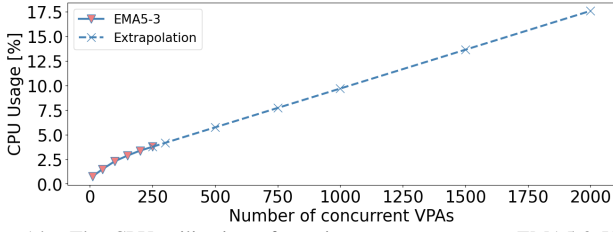


Fig. 14. The CPU utilization of running many concurrent EMA5-3 VPAs in Kubernetes. Because Kubernetes cannot run more than 250 VPAs, we extrapolate the CPU usage up to 2000 instances.

bounds, instead of under-estimating. Insufficient resources will lead to throttling of the applications. Conversely, offering many more resources than needed will lead to the under-utilization of the server operated by the cloud provider.

As our experiments show, the SMA and EMA-based approaches outperform both the ML-based autoscaling, as well as the default Kubernetes VPA. We achieve this result using two techniques. First, we adopt a *bottoming* mechanism for the future CPU demands prediction to prevent it from a rapid decrease. Second, we apply the same converging lower bound and upper bound as the default VPA. The lightweight SMA and EMA methods are more appropriate for serverless functions not only in cold invokes but also in warm runs.

### 3. Is dynamic CPU allocation feasible for serverless?

Currently, serverless providers offer clients only statically allocated CPU for running serverless functions. In this work we have investigated whether it is feasible to dynamically allocate CPU to serverless workloads, using tiny autoscalers. Based on our results, we are confident that simple and lightweight techniques, such as SMA and EMA are accurate in predicting and following the CPU utilization of serverless functions. Moreover, practitioners can run thousands of these tiny autoscalers without much CPU overhead. Implementing these in practice might offer serverless platforms the ability to better allocate resources to their clients by reducing overall server under-utilization and by reducing the throttling created by statically allocating CPU.

## VI. RELATED WORK

We discuss related work in four categories: prediction of workloads, autoscaling in the cloud, autoscaling containers, and vertical autoscaling virtual machines.

**Predicting Workload Trends.** Recent work [25] on CPU usage prediction for autoscaling is based on Holt-Winters exponential smoothing (HW) and Long Short-Term Memory (LSTM) methods. These two methods exhibit expensive computational complexity. We have analyzed them in detail in Section III. The lightweight prediction methods we modify in this article are based on the two-step CPU usage prediction model [24]. Similar to these techniques, Casolari and Andreolini [39] proposed another trend-aware regression model using linear extrapolation. Regression-based methods [40] also exist in this space, as well as techniques based on autoregression, introduced by Roy et al. [41].

**Cloud Autoscaling.** A complete taxonomy of the field of cloud autoscaling are developed by Chen et al. [42]. Going in

depth, several projects specifically investigate the performance study of the state-of-art autoscalers. Versluis et al. [23], Ilyushkin et al. [43] and Jindal et al. [44] demonstrate in-depth comparisons for autoscalers on workflows and introduce frameworks and tools to assess the performance of autoscalers. Efficient techniques [45] for achieving autoscaling include task allocation strategies, e.g., as the one introduced by Zhong and Buyya [46] or by Thurgood and Lennon [47], who offered a software solution to autoscale entire Kubernetes clusters. Recently, also serverless clouds have been the subject of autoscaling via reinforcement learning [48]. However, this only autoscales horizontally the number of container running hosts, not the containers themselves.

**Autoscaling Containers.** Rattihalli et al. [49] designed RUBAS, an autoscaling mechanism to estimate the CPU and memory resources of containers through the sum of the median of observations and the absolute deviation of observations [50], [51]. Autopilot [29] introduced by Google is a complete autoscaling system, it not only takes into account vertical autoscaling but also horizontal autoscaling on both CPU and memory usage. The current vertical pod autoscaling recommender we compare against in this article is directly inspired by the moving window recommenders in Autopilot. Nguyen et al. [52] investigated the horizontal pod autoscaling and offered optimization strategies for horizontal pod autoscaling.

**VM Vertical Autoscaling.** Similar techniques have long been applied to virtual machines in the cloud. These can be vertically either through hotplugging (for CPU or memory) or through CPU throttling (e.g., using rate limiting mechanisms). For example, several articles consider vertical VM autoscaling [53]–[56].

As opposed to all these, in this article we assess the feasibility of tiny autoscalers for short-lived serverless functions. Our results show that for these types of workloads, a special kind of autoscaler is needed, namely a lightweight autoscaler that is able to react quickly to changes in demand.

## VII. CONCLUSIONS

In this paper, we have addressed the problem of dynamically allocating CPU for serverless functions through tiny autoscalers. In modern serverless clouds, users request serverless functions of a certain memory size allocation. Subsequently, their CPU is allocated by the cloud provider proportionally to the memory allocation. It is therefore non-trivial for the user to achieve a satisfactory *CPU and memory allocation*. To solve this problem and scale CPU and memory independently, the literature offers various autoscaling algorithms. However, most of them are either heavy-weight or depend on expensive historical information, which may not be available for the short-running and infrequently invoked serverless functions. We therefore investigate five different algorithms for the dynamic rightsizing of serverless functions' CPU during runtime. We implement several of these in Kubernetes and experiment with state-of-the-art serverless workloads. We show that dynamic CPU rightsizing is possible for serverless functions and several

algorithms achieve good performance for both cold invokes as well as warm runs. Tiny autoscalers can be found on github: <https://github.com/ZhaoNeil/On-Demand-Resizing.git>

## ACKNOWLEDGMENTS

The work in this article was in part supported by The Dutch National Science Foundation NWO Veni grant VI.202.195.

## REFERENCES

- [1] Carreira *et al.*, “Cirrus: A serverless framework for end-to-end ml workflows,” in *SoCC*, 2019.
- [2] Jiang *et al.*, “Towards demystifying serverless machine learning training,” in *SIGMOD*, 2021.
- [3] Jonas *et al.*, “Occupy the cloud: Distributed computing for the 99%,” in *SoCC*, 2017.
- [4] Fouladi *et al.*, “Encoding, fast and slow: Low-latency video processing using thousands of tiny threads,” in *NSDI*, 2017.
- [5] Pu *et al.*, “Shuffling, fast and slow: Scalable analytics on serverless infrastructure,” in *NSDI*, 2019.
- [6] L. Toader, A. Uta, A. Musaafir, and A. Iosup, “Graphless: Toward serverless graph processing,” in *2019 18th International Symposium on Parallel and Distributed Computing (ISPD)*. IEEE, 2019.
- [7] Müller *et al.*, “Lambada: Interactive data analytics on cold data using serverless cloud infrastructure,” in *SIGMOD*, 2020.
- [8] Randazzo and Tinnirello, “Kata containers: An emerging architecture for enabling mec services in fast and secure way,” in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019.
- [9] Agache *et al.*, “Firecracker: Lightweight virtualization for serverless applications,” in *USENIX NSDI*, 2020.
- [10] Gunasekaran *et al.*, “Fifer: Tackling resource underutilization in the serverless era,” in *Proceedings of the 21st International Middleware Conference*, 2020.
- [11] Bhasi *et al.*, “Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms,” in *SoCC*, 2021.
- [12] AWS Lambda, “Aws lambda documentation,” Available at <https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html> (2021/12/10).
- [13] Shahrad *et al.*, “Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider,” in *USENIX ATC*, 2020.
- [14] Ustiugov *et al.*, “Benchmarking, analysis, and optimization of serverless function snapshots,” in *ASPLOS*, 2021.
- [15] Eismann *et al.*, “Sizeless: Predicting the optimal size of serverless functions,” *arXiv preprint arXiv:2010.15162*, 2020.
- [16] Ali-Eldin *et al.*, “An adaptive hybrid elasticity controller for cloud infrastructures,” in *Network Operations and Management Symposium (NOMS)*, 2012 IEEE, 2012.
- [17] Iqbal *et al.*, “Adaptive resource provisioning for read intensive multi-tier applications in the cloud,” *FGCS*, vol. 27, no. 6, 2011.
- [18] Urgaonkar *et al.*, “Agile dynamic provisioning of multi-tier internet applications,” *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, no. 1, 2008.
- [19] Fernandez *et al.*, “Autoscaling web applications in heterogeneous cloud infrastructures,” in *IC2E*, 2014.
- [20] Chieu *et al.*, “Dynamic scaling of web applications in a virtualized cloud computing environment,” in *E-Business Engineering, 2009. ICEBE’09. IEEE International Conference on*, 2009.
- [21] Ilyushkin *et al.*, “An experimental performance evaluation of autoscaling policies for complex workflows,” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017.
- [22] Bauer *et al.*, “Chamulteon: Coordinated auto-scaling of micro-services,” in *ICDCS*, 2019.
- [23] Versluis *et al.*, “A trace-based performance study of autoscaling workloads of workflows in datacenters,” in *CCGrid*, 2018.
- [24] Andreolini and Casolari, “Load prediction models in web-based systems,” in *Proceedings of the 1st International Conference on Performance Evaluation Methodologies and Tools*, 2006.
- [25] Wang *et al.*, “Predicting cpu usage for proactive autoscaling,” in *Proceedings of the 1st Workshop on Machine Learning and Systems*, 2021.
- [26] Pan, *Holt–Winters Exponential Smoothing*. American Cancer Society, 2011.
- [27] Hochreiter and Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, 1997.
- [28] Burns *et al.*, “Borg, omega, and kubernetes,” *ACM Queue*, vol. 14, 2016.
- [29] Rzadca *et al.*, “Autopilot: Workload autoscaling at google scale,” in *EuroSys*, 2020.
- [30] T. Sebastian, “Vertical pod autoscaler,” Available at <https://banzaicloud.com/blog/k8s-vertical-pod-autoscaler/> (2021/12/10).
- [31] 2021 The Kubernetes Authors, “Kubernetes monitoring architecture,” Available at [https://github.com/kubernetes/design-proposals-archive/blob/main/instrumentation/monitoring\\_architecture.md](https://github.com/kubernetes/design-proposals-archive/blob/main/instrumentation/monitoring_architecture.md) (2021/12/10).
- [32] Google, “cadvisor,” Available at <https://github.com/google/cadvisor> (2021/12/10).
- [33] 2021 The Kubernetes Authors, “Kubelet,” Available at <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/> (2021/12/10).
- [34] 2021 The Kubernetes Authorss, “Kubernetes metrics server,” Available at <https://github.com/kubernetes-sigs/metrics-server> (2021/12/10).
- [35] 2021 The Kubernetes Authors, “Metrics-api,” Available at <https://github.com/kubernetes/metrics> (2021/12/10).
- [36] 2021 The Linux Foundation, “What is prometheus?” Available at <https://prometheus.io/docs/introduction/overview/> (2021/12/10).
- [37] V. Kulkarni, “In-place update of pod resources,” Available at <https://github.com/kubernetes/enhancements/issues/1287> (2021/12/10).
- [38] Cooper *et al.*, “Benchmarking cloud serving systems with ycsb.” Association for Computing Machinery, 2010.
- [39] Casolari *et al.*, “Runtime prediction models for web-based system resources,” in *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, 2008.
- [40] Davis *et al.*, “Regression-based utilization prediction algorithms: An empirical investigation,” in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, 2013.
- [41] Roy *et al.*, “Efficient autoscaling in the cloud using predictive models for workload forecasting,” in *CLOUD*, 2011.
- [42] Chen *et al.*, “A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems,” *ACM Comput. Surv.*, vol. 51, no. 3, 2018.
- [43] Ilyushkin *et al.*, “An experimental performance evaluation of autoscalers for complex workflows,” *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, 2018.
- [44] Jindal *et al.*, “Autoscaling performance measurement tool,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018.
- [45] Ghanbari *et al.*, “Optimal autoscaling in a iaas cloud,” in *Proceedings of the 9th International Conference on Autonomic Computing*, 2012.
- [46] Zhong and Buyya, “A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources,” *ACM Trans. Internet Technol.*, vol. 20, no. 2, 2020.
- [47] Thurgood and Lennon, “Cloud computing with kubernetes cluster elastic scaling,” in *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*, 2019.
- [48] Schuler *et al.*, “Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments,” in *2021 IEEE/ACM CCGrid*. IEEE, 2021.
- [49] Rattihalli *et al.*, “Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes,” in *CLOUD*, 2019.
- [50] —, “Two stage cluster for resource optimization with apache mesos,” *CoRR*, vol. abs/1905.09166, 2019.
- [51] Rattihalli, “Exploring potential for resource request right-sizing via estimation and container migration in apache mesos,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018.
- [52] Nguyen *et al.*, “Horizontal pod autoscaling in kubernetes for elastic container orchestration,” *Sensors*, vol. 20, no. 16, 2020.
- [53] Hummaida *et al.*, “Adaptation in cloud resource configuration: A survey,” *J. Cloud Comput.*, vol. 5, no. 1, 2016.
- [54] Sedaghat *et al.*, “A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling,” in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, 2013.
- [55] Lu *et al.*, “Application-driven dynamic vertical scaling of virtual machines in resource pools,” in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–9.
- [56] Svärd *et al.*, “Hecatichire: Towards multi-host virtual machines by server disaggregation,” in *European Conference on Parallel Processing*. Springer, 2014, pp. 519–529.