ElaX: Provisioning Resource Elastically for Containerized Online Cloud Services

Yanan Yang, Laiping Zhao*, Zhigang Li, Lihai Nie, Peiqi Chen, and Keqiu Li
Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University, China

***Abstract*—To reduce the cost of online cloud services, service providers often employ the elastic approach that allows tenants to "scale out" or "scale up" their applications at runtime. However, the traditional virtual machine-based approach cannot meet the fine-grained fluctuating demand due to its slow startup time and high reconfiguration cost. To address this challenge, we present *ElaX*, an online service manager that minimizes the resource provisioning cost for containerized online services while guaranteeing their tail latency requirement. *ElaX* designs a workload-aware resource allocation mechanism for containerized online services through the collaboration of three key components: First, *workload predictor* is able to precisely predict the workload in periodic scenario, through a LSTM (Long Short-Term Memory) network; Second, *resource reservation* allocates the just-right amount of resource supporting the predicted workload, using the combination of both scale-up and scale-out operations; Third, *online controller* guarantees the tail latency requirement through a feedback-based control method, and further reduces the provisioning cost through resource reclamation. Our experiments on the two production workloads demonstrate that, when compared with existing methods, *ElaX* can reduce the average resource over-provisioning cost by more than 32.6% while guaranteeing the tail latency requirement.**

***Index Terms*—Cloud Computing, Resource provisioning, Tail Latency, Containers**

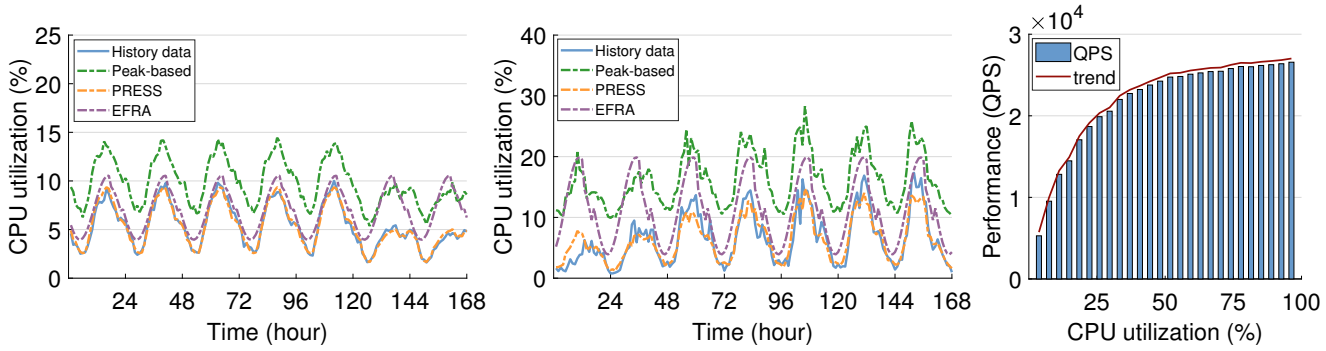
I. Introduction

Cloud computing frees service developers from complex and complicated maintenance work on hardware infrastructure, through simply renting the needed computing capacities (e.g., servers, storage, network.) from cloud providers, and paying for resource on demand. While users would always like to cut the cost of renting through precise shaping of resource requirements of their services, cloud providers cannot offer stable quality of services (QoS) due to the constantly changing workload [4] and unpredictable resource contention from the tenants sharing the cloud [6, 13, 21]. Hurting user experience is rather costly, for example, just one-second slowdown of page loading could cost $1.6 billion in sales of Amazon [10]. In this case, users have to resort to resource over-provisioning for guaranteeing their QoS. However, wasteful over-provisioning results in low resource utilization, thereby increasing the cost of cloud services. For example, resource reservations by Twitter could reach up to 80% of total capacity, while their production cluster's CPU utilization is constantly below 20% [7]. Similarly, traces from both Google [28] and Aliyun [23] showed that they merely achieve aggregate CPU utilization of 25-35% and aggregate memory utilization of 40%.

How to reduce the resource provisioning cost while guaranteeing the QoS is a significant challenge. To address this challenge, the existing work [12, 30] have been able to scale service cluster through increasing or decreasing the number of virtual machines (VMs), according to the fluctuating workload. However, workload-aware resource scaling systems [20, 31] support resource scaling only for batch jobs, and can not be directly applied to resource allocation for online service due to the long-term running feature. Even for online services, the VM-based scaling approaches are rather costly, since starting or reconfiguring a VM usually takes minutes, which is far longer than the millisecond-level tail latency SLO (service level objective) [5, 19]. If the workload towards a service changes significantly in a short time, a reactive VM scaling operation could be too slow to take effect. Fortunately, the advent of the lightweight container model [9] enables runtime resource reconfiguration in a small-time granularity, making scaling much simple and easy. *EFRA* [4] supports resource scaling-up for container-enabled cloud systems. However, their solution can only applicable in workload with strong stable periodic features, and does not support the flexible scaling-out and scaling-up combined decision. In addition, while Kubernetes [2] can both scale-out or scale-up the service cluster, its feedback-based approach cannot strictly guarantee the SLO.

In this work, we aim to further reduce the resource provisioning cost for long-running online services with the tail latency SLO guarantee. Since reducing the provisioned resources highly risk SLO violations, when and how many resources to scale should be very carefully decided. We suggest to scale the allocated resources both horizontally (scale-out) and vertically (scale-up), depending on the workload and cloud system status. There are several challenges towards this goal. First, although the production workload has shown periodic features, their periods are not always stable. Noises, like incremental periodical load, bursts and weekend drop, severely increases the prediction error, making the workload prediction extremely difficult. Second, given the workload estimates, how to derive the just-right amount of resources supporting this workload is also not easy. In particular, although the container technique provides a lightweight way for resource scaling, scaling-up and scaling-out still perform differently on startup costs. The optimal combination of scale-up and scale-out should be derived. Third, as the prediction error is inevitable,

*Corresponding author: laiping@tju.edu.cn



(a) Comparison of prediction results with ClarkNet (b) Comparison of prediction results with Calgary (c) Resource-Performance bottleneck

Fig. 1: The resource reservation by Peak, PRESS, EFRA under ClarkNet (a) and Calgary traces (b); The resource-performance bottleneck in Redis service node (c).

how to guarantee the tail latency SLO under prediction error is also a challenge.

We present *ElaX*, a resource scaling engine that minimizes the resource provisioning cost for containerized online services while guaranteeing the tail latency SLO. *ElaX* reduces the resource over-provisioning cost through a workload-aware resource scaling method. Compared with existing systems [4, 12, 30], *ElaX* improves the prediction accuracy in non-stable periodic workload and supports the combined scaling-up and scaling-out operation for minimizing the startup cost. It also integrates the feedback-based QoS management policy to avoid the possible SLO violations under prediction error. Our contributions can be summarized as follows:

1) To improve the prediction accuracy under non-stable workload, we filter out the noises in the historical workload data series using the SSA (Singular Spectrum Analysis) method [26], and utilize a deep learning method to predict the future workload. The evaluation result shows that our method can improve the prediction accuracy significantly, especially for the non-stable periodical workload.
2) We build a resource-performance model to characterize the relation between resource allocation and service throughput, directing the resource requirement for supporting the predicted workload. Then, we consider the resource-performance bottleneck and the heterogeneity of servers and derive the optimal combination of scaling-out and scaling-up operations for resource reservation.
3) We employ a feedback-based online controller to avoid the possible SLO violations due to the prediction error and present a resource reclamation mechanism to re-allocate the over-provisioned resources at a small-time granularity.
4) We implement *ElaX* based on Docker engine [3] and evaluate its efficiency in *redis* cluster and *e-commerce* with two different production workloads. Experimental results show that, when compared with existing four methods, *ElaX* reduces the average resource over-provisioning cost by >32.6% while guaranteeing the tail latency SLO.

II. Motivation

The periodic feature of access load towards an online service has been detected and confirmed repeatedly [4]. It is straightforward to reduce the over-provisioning cost through a workload-aware resource allocation method. However, previous work either is rather conservative in prediction, leaving a significant space for further reducing the cost, or underestimates the workload, resulting in SLO violations. We deploy an online service of *redis* [27] cluster, and experimentally evaluate the four existing resource scaling methods (including *Peak* [11], *PRESS* [12], *EFRA* [4] and *Kubernetes* [2]) using two production workloads *ClarkNet* and *Calgary* [18] as the request workload modes. The results are showed in Table I.

TABLE I: Comparison of existing methods

Method	Prediction Error		SLO	Scale	
	ClarkNet	Calgary	Guarantee	up	out
Peak	82%	148%	Yes	Yes	No
PRESS	5%	18%	No	Yes	No
EFRA	37%	73%	Yes	Yes	No
Kubernetes	–	–	No	Yes	Yes
ElaX	6%	15%	Yes	Yes	Yes

Peak [11] always chooses the maximum workload from the corresponding point in each historical period as the predicted value. Although this conservative policy can be very safe in enforcing the tail latency SLO, it wastes a large number of resources. Fig. 1(a) and Fig. 1(b) show that it provisions 75% and 124% more resources than the actual demands in *ClarkNet* and *Calgary* cases, respectively.

PRESS [12] derives the mean value of the corresponding points in each historical period as the predicted workload. It decreases the prediction error to 5% and 18% in *ClarkNet* and *Calgary* respectively. However, we find that this way always underestimates the actual resource demand and cannot strictly guarantee the SLO (see section IV).

EFRA [4] derives the "best-fit" period pattern based on the historical data and predicts the resource usage through a collaborative filtering-based recommendation method. It can

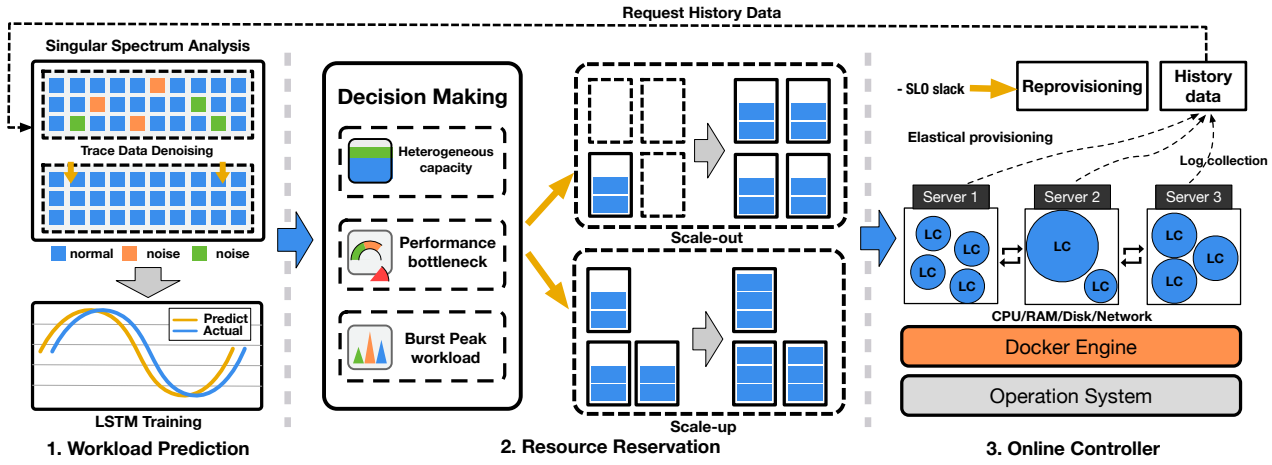


Fig. 2: ElaX System Architecture

achieve a very low prediction error only when the periodic feature is stable. However, the periodic feature may vary over time. For example, the seven days trace for *ClarkNet* shows a significant drop in the weekends (Fig. 1(a)), while the actual resource requirements in each period of *Calgary* continuously increase from Monday to Sunday (Fig. 1(b)). These trends do not affect the period length, but the actual resource requirements are different among periods. Thus, the overall prediction error by *EFRA* could achieve as high as 73% in *Calgary* case.

Kubernetes [2] supports both scale-out and scale-up operations through the HPA (Horizontal Pod Auto-scaling) and VPA (Vertical Pod Auto-scaling) techniques, respectively. However, they make resource scaling decisions only relying on the passive feedback from QoS monitoring, and does not support the workload prediction.

Workload-aware resource scaling engine heavily relies on the accuracy of workload prediction to reduce the resource over-provisioning cost. However, there still lacks a precise predictor that works well in unstable workload environment. In addition, the scale-up and scale-out operations may generate completely different performance improvement effects. Even for the same operation (either scale-up or scale-out), its impact on QoS also varies over the intensity of workload. Fig. 1(c) shows that when the CPU utilization exceeds 40%, the scale-up operation will not improve the throughput of *redis* quickly any more. That is, CPU is no longer the bottleneck resource for *redis* at this time, and it is better to employ the scale-out operation to support the processing of more requests in this case. Hence, the resource scaling engine should launch the scale-out and scale-up operation in a combined consideration based on the resource status and precise workload prediction.

III. ElaX Design

In this section, we present our design of *ElaX* and show how *ElaX* can reduce the resource over-provisioning cost based on the workload prediction.

A. Overview

Fig. 2 shows the system architecture of *ElaX*. It is designed to operate in a container-enabled system where each service instance runs as a container. We choose the container instead of virtual machine because it enables the resource adjustment operations without stopping and restarting the container, and the operation can take effect in tens of milliseconds. *ElaX* consists of three components as follows:

Workload predictor learns from historical workload and makes prediction on the next workload. To improve the accuracy of prediction under unstable periodic workload, we preprocess the historical data using the SSA method, then train a LSTM (Long Short-Term Memory) [17] network for prediction.

Resource reservation builds a resource-performance model to estimate the required resources supporting the predicted workload. It formulates the resource scaling as a mathematical programming problem considering the operation cost of scale-up and scale-out. Then, it derives the optimal scale-up and scale-out combination with minimum cost.

Online controller dynamically adjusts the allocated resources for the service during runtime. It is activated whenever the tail latency is close to SLO violation. Meanwhile, when the SLO returns to the safe level, we also design a resource reclamation algorithm to recycle the unnecessary resources for higher resource efficiency. The controller is implemented based on Linux's *cgroups*.

For *workload predictor*, since training a LSTM network takes time (around 30 minutes) and the model is usually stable enough within one period (e.g., one day in production environment), we just update the LSTM network once a day (at midnight). Then, the *workload predictor* runs once an hour, for predicting the workload per minute in the next hour (i.e., 60 points). Given the maximum predicted workload in next hour and the current system status, *resource reservation* also runs once per hour for reconfiguring the containers. We do not update container configurations every minute due to its

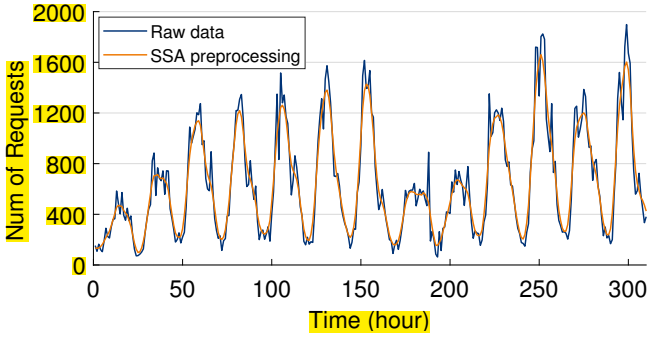


Fig. 3: Workload data series after processing with SSA. The blue line represents the original data series which contains a large number of noises, and the orange line represents the processed data using SSA, which is smoother than before. (Requests are collected by hours).

high cost. Finally, *online controller* continuously monitors the SLO and activates the resource adjustment whenever a SLO violation tends to occur.

B. Workload Predictor

1) Preprocessing: The production workload clearly shows periodic patterns in Fig. 1. However, these patterns are often unstable: Noises like weekend drop, incremental workload, burst peaks make the traditional prediction methods (e.g., exponential smoothing, optimal fitting, etc.) do not work. To address this challenge, we adopt the deep learning algorithm and train a LSTM network model to precisely predict the future workload.

Before the prediction, we process the historical workload data using the SSA method to filter out the local short-term noises. SSA [14] is commonly used to analyze one-dimensional time series data. It constructs a trajectory matrix according to the observed time series, and decompose it into a sum of components (e.g., long-term trend signal, periodic signal, noise signal) to analyze the structure of time series. Fig. 3 shows the reconstructed workload series after SSA processing, the processed data series is much smoother than before.

2) Prediction model: As the workload always changes in small-time granularity, single-point prediction generates frequent resource reconfigurations, leading to high cost. Hence, we instead predict multiple future points, for reducing the possible reconfiguration cost. For predicting the future workload, the existing literature is generally divided into two categories: statistical methods [15, 22] and machine learning methods [17]. We do not choose statistical methods, because they perform poorly for predicting multiple future points. While statistical methods, like Holt-Winters, may show high precision on single-point prediction, they predict the next token conditioned on its previously predicted token, and the approximation errors are iteratively accumulated.

Fig. 4 shows the prediction results by both LSTM and Holt-Winters, and Fig. 5(a) shows the accumulated errors by

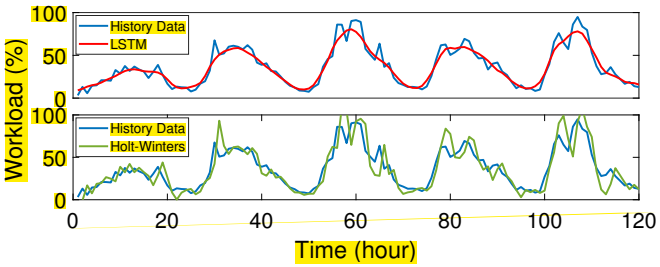
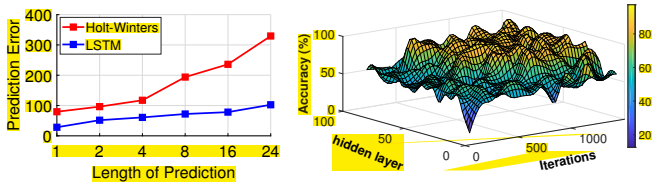


Fig. 4: LSTM vs. Holt-Winters

them. We see that LSTM performs much better accuracy than the latter with the prediction length increasing, so we choose LSTM as the prediction method in our system.



(a) LSTM vs. Holt-Winters (b) Parameter tuning

Fig. 5: Training the LSTM network model.

The structure of the LSTM leads to the significant difference in the above comparison result. It can update parameters automatically by propagated gradients and make a long-term prediction by building loss function on the entire generated workloads instead of every single one.

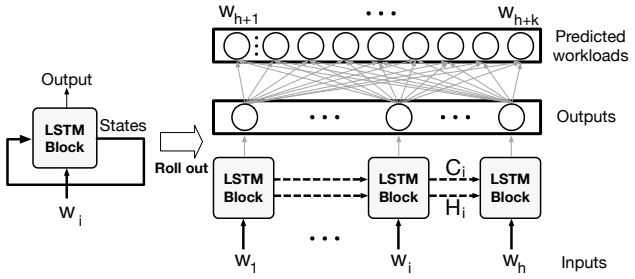


Fig. 6: LSTM Structure

Fig. 6 illustrates the design of LSTM network: The left shows a single LSTM block, which is trained recurrently by states and input data. While the generated states of a block and input data (i.e., w_i) are fed into the next block, the next block further computes an output and new states for the next. In this way, sequential features in historical data are maintained. Each state consists of two vectors: the block state vector c_i and the hidden state vector h_i. They are fed into the next block to initialize its corresponding states. Then, given the h outputs of LSTM blocks, we leverage a MLP (Multi-Layer Perceptron) network to generate k predictions denoting the k continuous workloads in the next period. That is, we have

$$(w_{h+1}, ...w_{h+k-1}, w_{h+k}) = LSTM(w_1, ..., w_{h-1}, w_h) \quad (1)$$

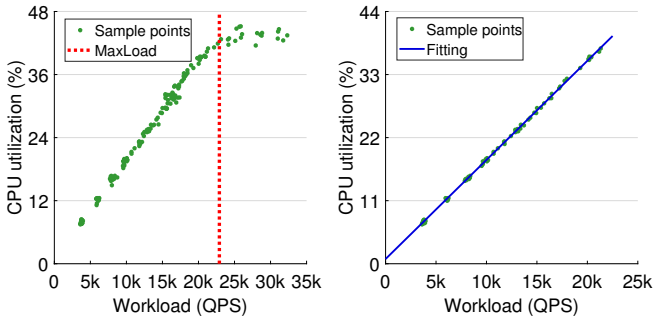
where w_i, $\forall i \in [1, .., h]$ is the historical workload at time i, and $w_j, \forall j \in [h+1, .., h+k]$ denotes the predicted workload at time j. For training the network, we record $(h+k)$ continuous points by sliding time windows on raw workload data. The first h points in each window are regarded as training data and the rest are treated as target data. Upon obtaining the training data, we use RMSE to measure the loss of predicted workloads and actual ones as:

$$\arg\min_{\theta} \sqrt{\frac{1}{k}\sum_{t=1}^{k}(p_t - w_t)^2} \tag{2}$$

where θ denotes the parameters set in LSTM model, and p_t and w_t are the predicted workloads and real value at time t respectively. For solving (2), we update θ using the common SGD (Stochastic Gradient Descent) method.

In our experiments, we conduct nearly 200 experiments to find the optimal parameter configurations with different combinations of neural layers and iterations (Fig. 5(b)). We finally get a well-trained LSTM model with 80 hidden layers and 520 iterations and the total training time is less than 30 minutes.

C. Resource reservation



(a) The actual CPU utilization under different workloads for Redis.

(b) Model linearly fitting.

Fig. 7: The resource-performance model.

1) Deriving the required resources: Given the predicted workload, we construct a resource-performance model to derive the amount of required resource. Fig. 7(a) shows the actual CPU utilization under different workloads for *redis*. Note that this analysis method can be applied in other online services. The CPU utilization and the workloads obviously have a non-linear relationship. We see that the CPU utilization increases linearly at the beginning, followed by a steady state. It means that increasing the CPU resources would improve the performance at the beginning, but become ineffective after a knee point. We denote the QPS (Query Per Second) at the knee in this case as *maxload*. Since the CPU resource will not be the performance bottleneck anymore after workload exceeds $maxload$, we can just allocate the maximum allowable resources in case of $> maxload$, and a fine-grained resource reconfiguration process would reduce the service cost only when the actual workload is less than $maxload$.

Considering the case of $< maxload$, we fit a linear model to describe the relationship between resource allocation and workload (Fig. 7(b)):

$$R = \sigma \cdot y + \beta \tag{3}$$

where R represents the required resources, y denotes the workload, σ and β are the coefficients of this linear model.

To improve the accuracy of model fitting, we detect and remove outliers using the *Nearest Neighbor* approach: For each data point, we compute its distance to the *k*-th nearest neighbor. The points that have the largest distances are identified as outliers. After removing the outliers, we derive the linear model that leads to the minimum euclidean distance to sample points.

2) Scale up and scale out: Given the required resources (denoted by R), it is a big challenge to derive the right number of containers and their configurations based on the existing container configurations and available physical machine resources because the "scale up" and "scale out" perform significantly different operation cost. Starting a new container involves the operations includes creating a container (C_{create}), adding the container into cluster (C_{add}), configuring the load balancer ($C_{balance}$) and the forwarding cost ($C_{forward}$) generated by cache miss due to the re-balancing mechanism in the cluster. Hence, the operation cost of scale-out ($C_{scaleout}$) is defined as follows:

$$C_{scaleout} = C_{create} + C_{add} + C_{balance} + C_{forward} \tag{4}$$

On the other hand, the scale-up operation, i.e., updating the configuration of a container, only involves the operation cost on changing the number of CPU cores and memory that allocated to the container. So we define the operation cost of scale-up ($C_{scaleup}$) as follows:

$$C_{scaleup} = C_{cpu} + C_{memory} \tag{5}$$

where C_{cpu} and C_{memory} represent the operation cost on varying the CPU and memory configuration, respectively. Note that the operation cost for scale-in and scale-down can be derived in the same way, and in fact, they are the same as the scale-out and scale-up operations.

In general, we prefer scaling-up the container cluster to scaling-out operation for processing the increased workload, because $C_{scaleup} < C_{scaleout}$ is usually established for scale-out's more operations than scale-up. However, it is not possible to scale up resources unboundedly due to the limited capacity of the physical machine. Moreover, the dominant resource may change over the workload, and scaling up a container could become useless after the workload reaches *maxload*.

Suppose the service cluster has n physical machines and can hold at most m containers. Let y_i^0, $\forall i \in [1, ..m]$ be the resources initially allocated to container i, and H_j, $\forall j \in [1, .., n]$ be the maximum capacity of physical machine j. Then, given the resource requirement R, the resource capacity H_j, $\forall j \in [1, .., n]$ and y_i^0, $\forall i \in [1, ..n]$, our goal is to find the

new configurations for all containers: y_i, $\forall i \in [1, ..n]$, so that the overall operation costs are minimized:

$$C = N_{scaleup}C_{scaleup} + N_{scaleout}C_{scaleout} \quad (6)$$

where $N_{scaleup}$ denotes the number of containers whose resource configuration is changed, and $N_{scaleout}$ denotes the number of new container instances.

Besides the variable y_i, we also define two other variables to formalize the problem as a MINLP (Mixed Integer Nonlinear Programming) problem:

- x_{ij}: a binary variable, which has the value "1" if container i is assigned to machine j. Otherwise, it is set to "0".
- z_i: a binary variable, denoting whether there is a change on resource configuration for container i. It has the value "1" if $y_i \neq y_i^0$. Otherwise, it is set to "0".

Let x_{ij}^0 be the initial assignment of container i. If $\sum_j x_{ij}^0 = 0$, it means container i has not been created yet. Thus, we have,

$$N_{scaleout} = |\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} - \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}^0| \quad (7)$$

$$N_{scaleup} = \sum_{i=1}^{m} z_i \quad (8)$$

Objective:

$$minimize: \quad C \quad (9)$$

$$\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}y_i = R, \quad (10)$$

$$\sum_{i=1}^{m} x_{ij}y_i \leq H_j, \quad \forall j \in [1, .., n] \quad (11)$$

$$(1 - z_i)(y_i - y_i^0) = 0, \quad \forall i \in [1, .., m] \quad (12)$$

$$(1 - z_i) + (y_i - y_i^0)^2 > 0, \quad \forall i \in [1, .., m] \quad (13)$$

$$(1 - \sum_{j=1}^{n} x_{ij})y_i = 0, \quad \forall i \in [1, .., m] \quad (14)$$

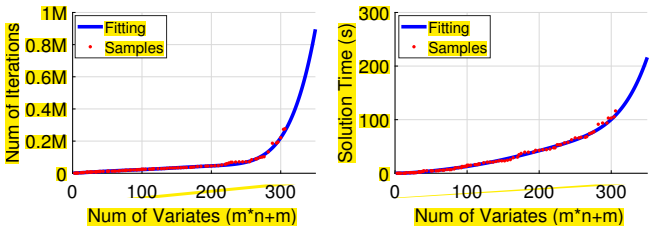
$$(1 - \sum_{j=1}^{n} x_{ij}) + y_i > 0, \quad \forall i \in [1, .., m] \quad (15)$$

$$\sum_{j=0}^{n} x_{ij} \leq 1, \quad \forall i \in [1, .., m] \quad (16)$$

$$x_{ij}, z_i \in \{0, 1\}, \quad \forall i, \forall j \quad (17)$$

$$0 \leq y_i \leq \overline{y}, \quad \forall i \in [1, .., m] \quad (18)$$

- Constraint (10) ensures that the required resources are fully satisfied.
- Constraint (11) makes sure that the aggregated resources allocated to containers deployed on a machine do not exceed the capacity of the machine.
- Constraint (12), (13) together with constraint (17) make sure that either $z_i = 1$ if and only if $y_i \neq y_i^0$ or $z_i = 0$ if and only if $y_i = y_i^0$.
- Constraint (14), (15) together with constraint (17) make sure that when container i has been created, its configuration y_i should not be "0". Otherwise, it is set to "0".
- Constraint (16) enforces that each container can only be deployed on at most one physical machine.
- Constraint (17) and (18) refer to the domain constraints.



(a) Iterations (b) Time

Fig. 8: The number of iterations and solution time required for solving the MINLP.

We solve the above problem using LINGO and generate the configuration updating plan based on the current configurations and the derived solution. Fig. 8 shows the number of iterations and time for solving this MINLP. We see that the number of iterations grows exponential over the number of variables $m * n + m$, and the solution time approaches to 100 seconds when the number of variables is 300. As the container reconfiguration is activated once per hour, this cost is acceptable.

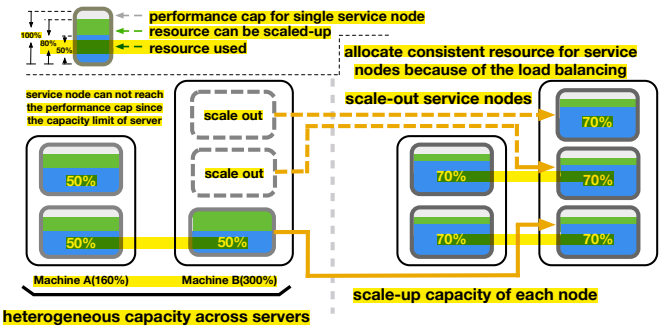
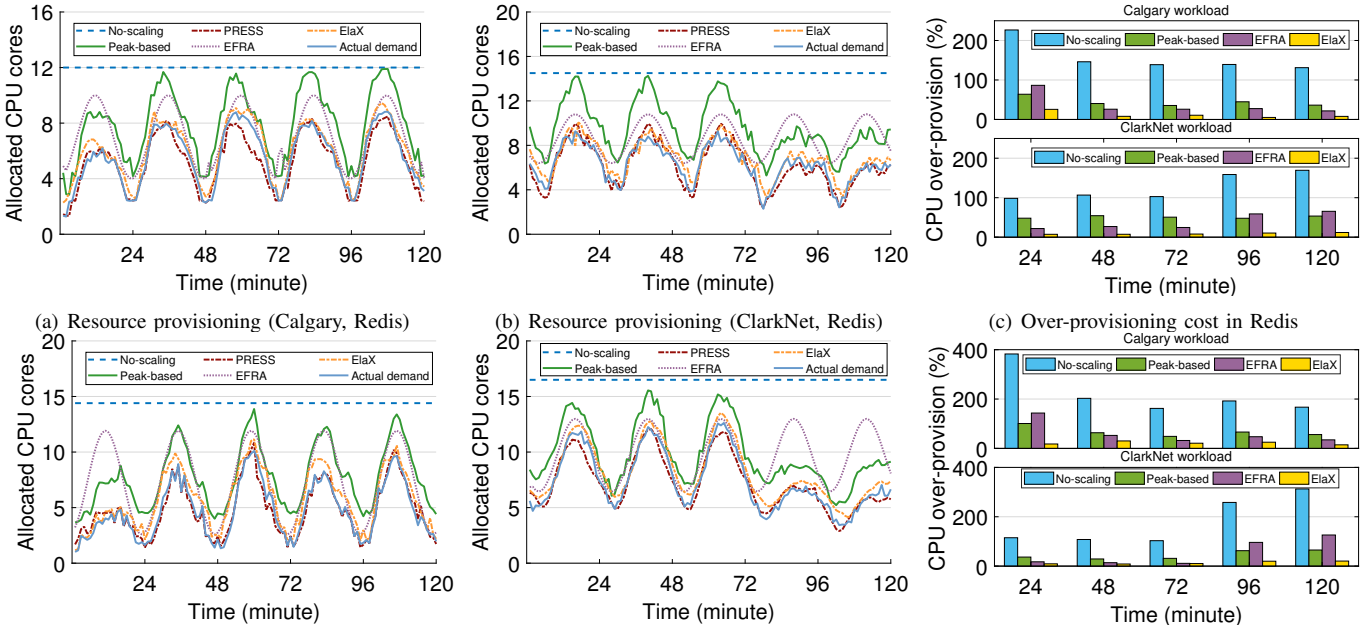


Fig. 9: When the workload increases from 150% to 350%, we scale out two container instances and scale up the resources of three existing containers from 50% to 70%.

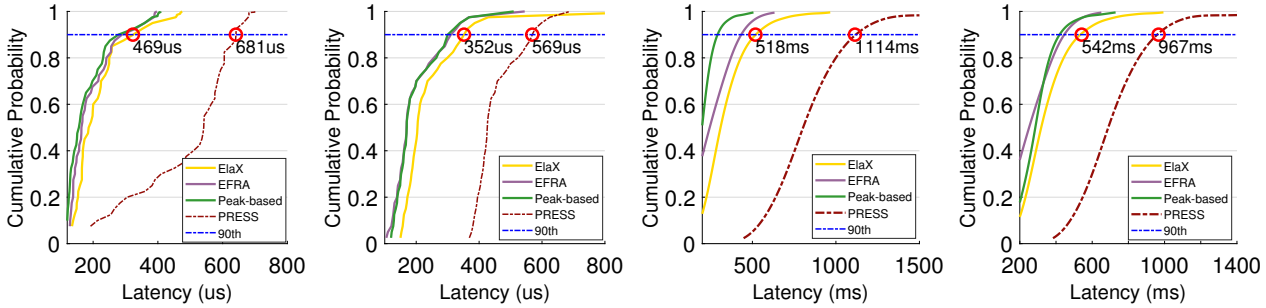
Fig. 9 shows an example of the resource scaling process of *ElaX*: machine A's resource capacity is 160% and holds two container instances, while machine B's capacity is 300% and there is only one container running on it at present. Each container has a performance cap of 100%. When the workload increases from 150% to 350%. The existing three active containers cannot satisfy the new requirement because the maximum capacity for them is only 260% (A can only reach the 80% of their performance cap because A only has 160% resource capacity, and the capacities of all containers are kept consistent for load balancing), so the cluster has to be extended to five nodes: while we scale up the existing three containers to 70% resources, we also scale out two new containers.



(a) Resource provisioning (Calgary, Redis) (b) Resource provisioning (ClarkNet, Redis) (c) Over-provisioning cost in Redis

(d) Resource provisioning (Calgary, e-commerce) (e) Resource provisioning (ClarkNet, e-commerce) (f) Over-provisioning cost in e-commerce

Fig. 10: Resource over-provisioning comparison of four methods (enabling only scale-up operation).



(a) Latency (Calgary, Redis) (b) Latency (ClarkNet,Redis) (c) Latency (Calgary, e-commerce) (d) Latency (ClarkNet, e-commerce)

Fig. 11: Tail latency CDF of four methods (SLO is set to $90th$ percentile latency $< 500us$ in Redis, $650ms$ in e-commerce).

D. Online Controller

The inevitable prediction error by *workload predictor* possibly leads to SLO violations. Therefore, we further design an *online controller* to adjust the resources at runtime according to the SLO-violation feedback from performance monitoring. It works as a daemon process and monitors the real-time latency of long running service, when the service performance declines or approaches a dangerous level, the resource reallocation mechanism will be activated and protect the service from violating the SLO.

The detailed control method is shown in Algorithm 1. We denote SLO_Target as the latency target of long running service, and it is set as the $90th$ percentile latency under the $maxload$ while with sufficient resource. By continuously monitoring the tail latency performance in consecutive 30-second windows, we derive a *slack* indicating the gap between current latency and SLO. If $slack < 0$, it means the SLO has been violated, and the system will allocate additional $0.1\times$ reserved resource for the service. If $0 < slack < 0.05$, it means the latency is closely approaching the SLO_Target and may soon violate the SLO. In this case, we allocate $0.05\times$ additional resource for preventing the possible violation of SLO. Otherwise, the latency is still safe enough for guaranteeing the SLO, we activate the *recycling* process, such that the over-allocated resources are recycled for other services. In lines 8-

Algorithm 1: Online control algorithm

```
1  while True do
2      slack = (SLO_Target − latency)/SLO_Target;
3      if slack < 0 then
4          increResource(curResource*0.1);
5      else if 0 < slack < 0.05 then
6          increResource(curResource*0.05);
7      else
8          extraResource = curResource − preResource;
9          if extraResource > 0 then
10             removeResource(extraResource*0.5);
11     sleep(2s);
```

10, we first derive the over-allocated resources (*extraResource*) using the currently allocated resources (*curResource*) and the predicted resources (*preResource*). Then, we recycle half of the over-allocated resource each time to reduce the waste.

IV. Performance Evaluation

A. Experiment Setup

1) Services: We deployed two online services *e-commerce website* and *redis*, considering their representativeness in service architecture and response time:

- *E-commerce* adopts the multi-tier architecture and usually responds in milliseconds. We use the TPC-W [24] as the transactional web e-commerce benchmark.
- *Redis* is an in-memory data structure store and is used as a database, cache and message broker. It adopts the fan-out architecture and usually responds in microseconds.

2) Workloads: We simulate the access workload towards the two services using production workload from ClarkNet and Calgary trace [18], by comparing *ElaX* with *EFRA*, *PRESS* and *Peak* methods, we evaluate the efficiency using resource utilization and $90th$ percentile latency.

3) System environment: We deploy the *redis* cluster with 22 service nodes and the *e-commerce* cluster with 20 nodes across four servers, and each container is initially allocated with 1 CPU cores and 2GB RAM. Clients are deployed in another server to avoid resource contention with service nodes. Each server is configured with 40 cores of 2.0 GHz Intel Xeon E7-4820 v4 and 128GB of DRAM. We use the operating system of Ubuntu 14.04 and the docker engine 18.03. All servers are connected to a 1000 Mbps switch. After deploying *ElaX* on this system, we measure that its average CPU utilization is $< 5\%$ and memory utilization is $< 400MB$.

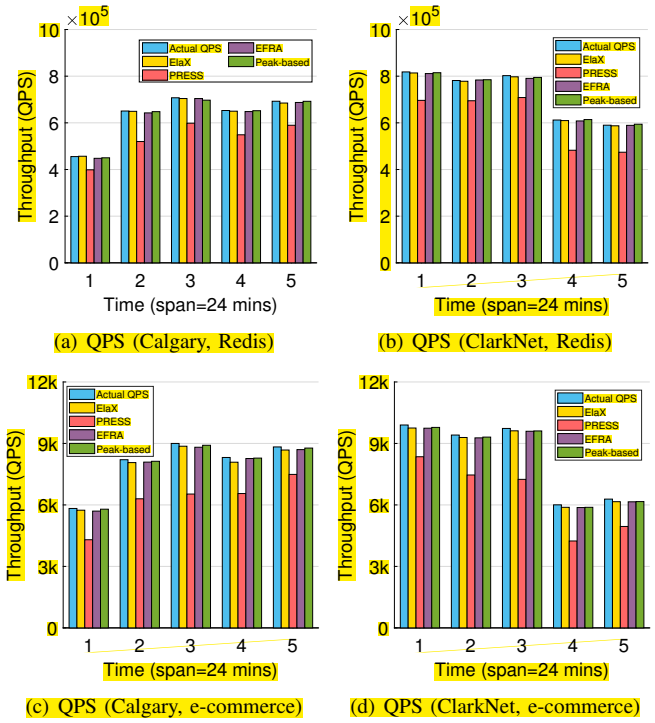
B. Results

1) Scale-up: Since the compared algorithms can only support scale-up operation, we evaluate *ElaX* by only enabling the scale-up operation at first.

Resource provision: Fig. 10 shows the allocated resources by the four algorithms. We only show the CPU resource here, but it can be extended to other resources easily. In the case of *redis* (Fig. 10(a)-Fig. 10(c)), although *PRESS* always utilizes the least resources (even less than the actual demand), it leads to very high tail latency and cannot guarantee the tail latency SLO (Fig. 11). Among *ElaX*, *Peak*, *EFRA* and *No-scaling*, we find that, *No scaling* wastes the largest amount of CPU (by an average of 141%) because of the fluctuating CPU demands of the workload. *Peak* can dynamically allocate CPU resource for the varying workloads, but it also causes a significant waste of 46% resources, since its conservative approach that uses worst-case peak demand as the predicted workload in next period. The over-provisioning cost by *EFRA* is around 34%, which is much less than *No Scaling* and *Peak*. However, it also exceeds 70% in the first period, since its prediction does not work well in unstable periodic workload. *ElaX* performs much better than the former algorithms, and its over-provisioning cost is generally less than 10% because of its high prediction accuracy.

In case of *e-commerce*, Fig. 10(d)-Fig. 10(f) shows the similar results. While *No-scaling*, *Peak* and *EFRA* averagely generate over-provisioning cost of 200.3%, and 29.6%, respectively, our *ElaX* only generates less than 18% of over-provisioning cost.

Tail latency: The tail latency SLO of online service must be guaranteed while allocating the resource elastically. Fig. 11 shows the CDF (Cumulative Distribution Function) of latency by four algorithms: *Peak*, *EFRA*, *PRESS* and *ElaX*. We set the tail latency SLO as the $90th$ percentile latency under the *maxload* while allocated with sufficient resources, which are $500us$ and $650ms$ for *redis* and *e-commerce*, respectively. We find that *PRESS* cannot guarantee the SLO in all cases: its $90th$ percentile latency for *redis* under *Calgary* and *ClarkNet* trace are 681us and 569us, respectively, and for *e-commerce* under *Calgary* and *ClarkNet* trace are 1114ms and 967ms, respectively. On the other hand, *Peak* and *EFRA* can guarantee the SLO because of its over-provisioned resources, and *ElaX* can guarantee the SLO because of its high prediction precision and online control mechanism.

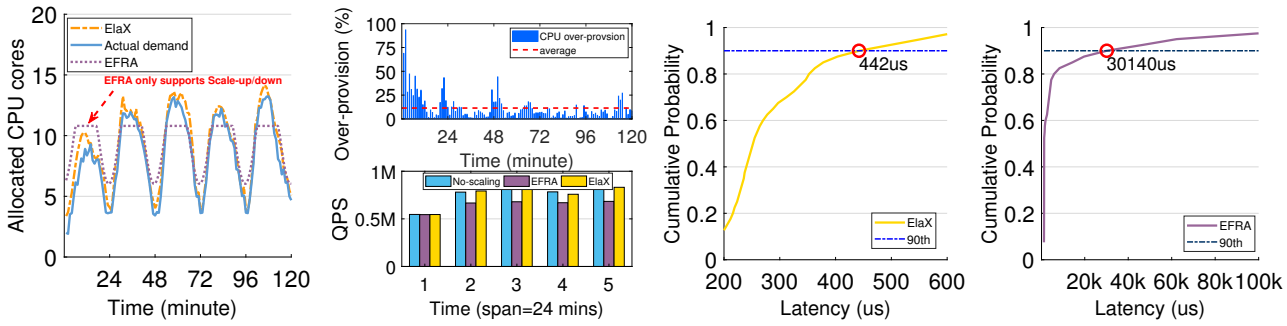


(a) QPS (Calgary, Redis) (b) QPS (ClarkNet, Redis)

(c) QPS (Calgary, e-commerce) (d) QPS (ClarkNet, e-commerce)

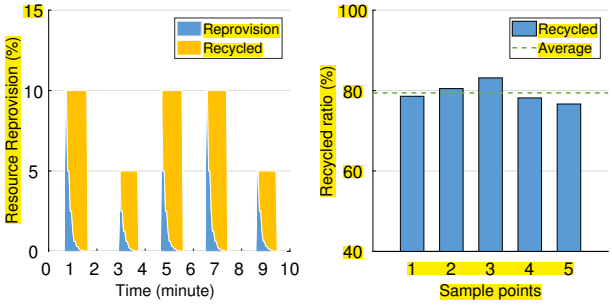
Fig. 12: System Throughput (scale-up).

Throughput: *ElaX* provides a highly precise prediction on workload and the amount of required resources. However, when we actually allocate the corresponding amount of resources, the actual system throughput (i.e., QPS) is slightly decreased. Fig. 12 shows the system throughput generated by the four methods. We see that *Peak*, *EFRA*, *ElaX* decrease the throughput by an average of 0.52%, 0.81% and 0.69%, respectively, all of them are less than 1%. The instability of



(a) Resource provisioning cost (b) Throughput (c) Latency of ElaX (d) Latency of EFRA

Fig. 13: Resource utilization of Redis under Calgary (enabling both scale-up and scale-out operations).



(a) Re-provisioning and recycling (b) Recycled resources

Fig. 14: Recycling resources.

the system may lead to fluctuations in QPS, so we suggest allocating slight more resources for the online service after we get the resource allocation using the resource-performance model. For *PRESS*, it cannot guarantee the throughput in all experimental groups.

2) Scale-up and scale-out: We change the system configurations by constraining the maximum usable capacity of two containers to 60% of its original capacity, that is, resulting in a heterogeneous system. We run *ElaX* on the heterogeneous system, and enable both scale-up and scale-out operations. We compare *ElaX* only with *EFRA* because *EFRA* performs the best among the other algorithms.

Resource provision: Fig. 13(a) shows allocated CPU resources by *EFRA* and *ElaX*. We see that the workload increases by almost 0.5% from the first time window to second window, *EFRA* can only provision resource by scale-up/down and cannot guarantee the increased workload in second timespan, while *ElaX* can continually provide resource for the workload using scale-up/out. The top figure in Fig. 13(b) shows the resource over-provisioning cost by *ElaX* over time. We see that the over-provisioning cost is rather high at the beginning, but it declines quickly due to the continuous reconfigurations. The average CPU over-provisioning cost is about 11.4%. For the system throughput, the bottom figure in Fig. 13(b) shows that *ElaX* can achieve comparable throughput as the *No-scaling* while *EFRA* reduces the throughput by around 12%. The limitation of *EFRA* leads to the lower throughput in this heterogeneous system.

Tail latency: Fig. 13(c) and Fig.13(d) show the tail latency generated by both *ElaX* and *EFRA*. We see that *ElaX* can guarantee the SLO with a $90th$ latency of $442us$. In case of *EFRA*, its $90th$ percentile latency even exceeds $30ms$, which is much larger than the SLO target, thus it cannot guarantee the tail latency SLO under unstable periodic workload.

Resource recycling: We also present the efficiency of resource recycling mechanism of *ElaX*. Fig. 14 shows the timeline of the resource re-provisioning process in our experiment. We see that, the re-provisioning mechanism is triggered for five times, and is followed by five times of recycling.

When *ElaX* detects that the current tail latency will soon approach SLO (i.e., $slack < 0.05$), it allocates additional 5% more resources. If SLO has been violated, it allocates additional 10% more resources to guarantee SLO. After the allocation of additional resources, the recycling algorithm recycles extra resources in a bisection way if the SLO is recovered. We highlight the amount of recycled resources in Fig. 14(a), and find that the recycling mechanism recycles 79% of resources averagely during this time (Fig. 14(b)).

V. Related Work

Cloud management framework: Many cloud resource management systems, such as Yarn [32], Borg [33], Omega [29], Mesos [16], have been proposed to improve the resource efficiency of cloud systems. However, they never focus on the long-running online services, and do not provide the efficient resource allocation for online services. Kubernetes [2] supports both scale-out and scale-up operations by the HPA and VPA techniques, according to the feedback from latency monitoring. However, as the online service's workload often shows periodic patterns, it is possible to further reduce the provisioning cost through workload-aware resource allocations.

Resource allocation: There has been some work supporting workload-aware resource allocation for batch work to achieve the SLO effect without over-provisioning. Morpheus [20] is able to automatically allocate resources learning from historical resource usage. TetriSched [31] estimates the job runtime for planning ahead for a busty preferred resource type. Quasar [7] increases resource utilization through a prediction on interference between applications. HCloud [8], a hybrid provisioning system that uses both reserved and on-demand resources in a combined way for reducing the service cost.

However, there is still a lack of study on the access load pattern of the online service, for further reducing the resource provisioning cost.

Long-term online service: For long-running services, PRESS [12] extracts fine-grained dynamic patterns from application resource requirements and automatically adjusts the allocation of their resources. CloudScale [30] uses an online resource demand forecasting scheme to achieve adaptive resource allocation. However, due to under-estimation, PRESS and CloudScale cannot strictly guarantee the tail latency SLO. AGILE [25] dynamically adjusts the number of VMs allocated to cloud applications to keep up with load changes. However, starting or stopping a VM (or server) usually takes a few minutes, and these methods are not sufficient to effectively cope with the uncertain needs of long-term operation, especially for online services [1]. EFRA [4] supports the fine-grained resource allocation with a workload-aware allocation mechanism, but their workload prediction is highly inaccurate in the very common unstable periodic workload scenarios.

VI. Conclusion and Future Work

In this paper, we design an elastic resource provisioning framework: *ElaX*, which can dynamically adjust the resource allocation for periodical workload while guarantee tail latency SLO for online service. *ElaX* can precisely predict the complex workload variations, and scale the resources efficiently using both scale-up and scale-out operations in a combined way. Experiment results demonstrate the efficiency of *ElaX*. In the future, we would like to further improve the resource efficiency employing the hardware and software interference isolation techniques.

References

[1] Omer Y Adam, Young Choon Lee, and Albert Y Zomaya. Constructing performance-predictable clusters with performance-varying resources of clouds. *IEEE Transactions on Computers*, (9):2709–2724, 2016.

[2] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, (3):81–84, 2014.

[3] Carl Boettiger. An introduction to docker for reproducible research. *Acm Sigops Operating Systems Review*, 49(1):71–79, 2015.

[4] Binlei Cai, Rongqi Zhang, Laiping Zhao, and Keqiu Li. Less provisioning: A fine-grained resource scaling engine for long-running services with tail latency guarantees. In *Proceedings of the 47th International Conference on Parallel Processing*, page 30. ACM, 2018.

[5] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Elastic virtual machine for fine-grained cloud resource provisioning. In *Global Trends in Computing and Communication Systems*, pages 11–25. Springer, 2012.

[6] Christina Delimitrou and Christos Kozyrakis. ibench: Quantifying interference for datacenter applications. In *2013 IEEE international symposium on workload characterization (IISWC)*, pages 23–33. IEEE, 2013.

[7] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[8] Christina Delimitrou and Christos Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. *SIGPLAN Not.*, 51(4):473–488, March 2016.

[9] Docker. What is a container?, 2019. https://www.docker.com/resources/what-container.

[10] Kit Eaton. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon16-billion-sales, 2018.

[11] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, Guillaume Belrose, Tom Turicchi, and Alfons Kemper. An integrated approach to resource pool management: Policies, efficiency and quality metrics. In *DSN*, pages 326–335. IEEE, 2008.

[12] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. *CNSM*, 10:9–16, 2010.

[13] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *SoCC*, pages 22:1–22:14, 2011.

[14] Hossein Hassani. Singular spectrum analysis: meth. and comp. 2007.

[15] Yan He, Ashley Flavel, Zihui Ge, Alexandre Gerber, Massey Dan, Christos Papadopoulos, Hiren Shah, and Jennifer Yates. Argus: End-to-end service anomaly detection and localization from an isp's point of view. In *IEEE Infocom*, 2012.

[16] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[17] S Hochreiter and J Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[18] InternetTrafficArchive. http://ita.ee.lbl.gov/html/traces.html, December 16 2018.

[19] Waheed Iqbal, Matthew N Dailey, and David Carrera. Sla-driven dynamic resource management for multi-tier web applications in a cloud. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 832–837. IEEE Computer Society, 2010.

[20] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[21] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *High PERFORMANCE Computing, Networking, Storage and Analysis*, pages 1–12, 2012.

[22] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247, 2003.

[23] Qixiao Liu and Zhibin Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace. In *SoCC*, pages 347–360, 2018.

[24] D. A. Menasce. Tpc-w: A benchmark for e-commerce. *IEEE Internet Computing*, 6:83–87, 05 2002.

[25] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, volume 13, pages 69–82, 2013.

[26] Vicente Oropeza and Mauricio Sacchi. Simultaneous seismic data denoising and reconstruction via multichannel singular spectrum analysis. *Geophysics*, 76(3):V25–V32, 2011.

[27] Redis. Redis: an open source, in-memory data structure store, 2018. https://redis.io.

[28] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[29] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Eurosys*, pages 351–364. ACM, 2013.

[30] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[31] Alexey Tumanov, Timothy Zhu, and Jun Woo et al. Park. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, pages 35:1–35:16, 2016.

[32] Vinod Kumar Vavilapalli, Arun C. Murthy, and Chris et al. Douglas. Apache hadoop yarn: Yet another resource negotiator. In *SoCC*, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[33] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.