The Power of Prediction: Microservice Auto Scaling via Workload Learning

Shutian Luo[*][‡]
Shenzhen Institute of Advanced Technology, CAS
Univ. of CAS, Univ. of Macau
st.luo@siat.ac.cn

Huanle Xu[*][‡]
University of Macau
huanlexu@um.edu.mo

Kejiang Ye[‡]
Shenzhen Institute of Advanced Technology, CAS
kj.ye@siat.ac.cn

Guoyao Xu
Alibaba Group
yao.xgy@alibaba-inc.com

Liping Zhang
Alibaba Group
liping.z@alibaba-inc.com

Guodong Yang
Alibaba Group
luren.ygd@taobao.com

Chengzhong Xu[†][‡]
University of Macau
czxu@um.edu.mo

ABSTRACT

When deploying microservices in production clusters, it is critical to automatically scale containers to improve cluster utilization and ensure service level agreements (SLA). Although reactive scaling approaches work well for monolithic architectures, they are not necessarily suitable for microservice frameworks due to the long delay caused by complex microservice call chains. In contrast, existing proactive approaches leverage end-to-end performance prediction for scaling, but cannot effectively handle microservice multiplexing and dynamic microservice dependencies.

In this paper, we present Madu, a proactive microservice auto-scaler that scales containers based on predictions for individual microservices. Madu learns workload uncertainty to handle the highly dynamic dependency between microservices. Additionally, Madu adopts OS-level metrics to optimize resource usage while maintaining good control over scaling overhead. Experiments on large-scale deployments of microservices in Alibaba clusters show that the overall prediction accuracy of Madu can reach as high as 92.3% on average, which is 13% higher than the state-of-the-art approaches. Furthermore, experiments running real-world microservice benchmarks in a local cluster of 20 servers show that Madu can reduce the overall resource usage by 1.7× compared to reactive solutions, while reducing end-to-end service latency by 50%.

[*]Co-first author. Both authors contributed equally to this paper.
[†]Corresponding author.
[‡]S. Luo, H. Xu, K. Ye and C. Xu are also with Guangdong-Hong Kong-Macao Joint Laboratory of Human-Machine Intelligence-Synergy Systems.

CCS CONCEPTS

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

KEYWORDS

Microservices, Proactive Auto-scaler, Workload Uncertainty Learning

ACM Reference Format:
Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The Power of Prediction: Microservice Auto Scaling via Workload Learning. In *Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3542929.3563477

1 INTRODUCTION

Cloud service providers like Google and Alibaba tend to provide off-the-shelf microservice architectures for users to build their applications [1, 22]. Under this kind of architecture, lightweight microservices are deployed alongside a large number of long-running containers. These containers are usually controlled by a container manager that can flexibly schedule and scale containers, such as Kubernetes [28] and Swarm [49]. Despite flexibility, today's microservice architecture often leads to low cluster utilization due to its over-provisioning of peak resource demands to satisfy service level agreements (SLA) [30].

To mitigate this problem, a general framework is to automatically scale the number of containers for each microservice as the workload changes over time using either reactive or proactive approaches. Reactive approaches tune resources according to the current workload and the performance feedback from the system; see [21, 26, 37, 43, 53, 56] for examples. In contrast, proactive approaches use prediction methods to estimate performance in advance and then choose the resource configuration that results in the smallest resource usage [12, 16, 20, 40, 41, 55].

Reactive approaches work well for monolithic architectures [17, 42, 44], because the overhead of coarse-grained scaling is often tolerable. They perform unsatisfactorily under microservice frameworks due to two key issues. First, the chain of microservice calls serving service requests can be very long. As reported from Alibaba traces, the longest microservice call chain contains more than one hundred microservices [31]. It takes a long time for the microservice at the bottom of the chain to experience workload changes. When such a microservice observes overload, it may be too late to scale. Second, scaling each microservice typically requires fetching container images from the repository before preparing the environment to run, which takes seconds to complete [12, 41]. However, SLAs of online services are often defined within several hundred milliseconds [55]. Therefore, reactive scaling can easily lead to SLA violations.

In comparison, current proactive approaches mainly focus on predicting the end-to-end performance of service requests based on microservice dependency graphs. These solutions do not take into account two distinct characteristics of microservices, namely dynamic dependencies and microservice multiplexing. First, requests from the same online service can go through different sets of microservices at runtime, resulting in dynamic graphs with significantly different topologies [31]. This highly dynamic nature makes graph-based training difficult to be used in practice. Second, a single microservice can be multiplexed among multiple online services. A characterization study of Alibaba microservice traces shows that 5% of microservices are shared by more than 90% of online services [31]. Training all graphs independently may not be able to deal with microservice multiplexing.

In this paper, we aim to take a proactive approach to microservice autoscaling by predicting the performance of each individual microservice rather than the entire microservice dependency graph. The ability to achieve accurate prediction is highly dependent on the knowledge of each microservice's workload. In literature, many works have investigated workload learning for monolithic cloud applications [13, 33, 34, 38, 44, 45]. They are mainly designed for independent workloads, hardly applicable for microservices. A microservice workload often exhibits strong data-dependent uncertainty in addition to periodical patterns. That is, peak workloads have much higher variance across different periods (relative to the mean workload) than other workloads due to the dynamic dependency between microservices. As a result, conventional prediction approaches for monolithic applications tend to make underestimations for microservices. For example, ARIMA model (autoregressive integrated moving average) [13, 15] could result in a 40% under-estimation at peak workloads, causing SLA violations.

In this paper, we design Madu, a $\underline{M}$icroservice $\underline{A}$uto-scaler that can efficiently handle $\underline{D}$ata-dependent $\underline{U}$ncertainty. A key observation behind Madu is that microservices have varying degrees of workload uncertainty. The smaller the in-degree of a microservice, the higher the uncertainty. Madu builds predictive models for each microservice independently. Under each model, Madu designs a new loss function to incorporate the effect of data-dependent uncertainty. By minimizing the loss function, Madu is able to simultaneously learn the mean and uncertainty of microservice workloads. Madu also combines these two terms elaborately to produce accurate predictions.

Another observation behind Madu is that the microservice workload per container is strongly correlated with the container's OS-level metrics (such as CPU and memory utilization), more than with application-level metrics (i.e., microservice response time). Madu adopts these OS-level metrics to design a container-based auto-scaler. Specifically, Madu dynamically scales the number of deployed containers per microservice based on the predicted microservice workload so that the CPU and memory utilization of each container reaches predefined thresholds. Furthermore, Madu runs convex optimization to avoid scaling a large number of containers when the workload fluctuates widely within a short period.

To demonstrate the effectiveness of our design, we first validated Madu's workload predictor using over 1000 microservices in Alibaba production clusters [1]. The evaluation results show that the predictor can achieve much higher prediction accuracy compared to ARIMA model [13, 15], *Seq2Seq*

[1]The microservices workload traces are now open for public access via https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022.

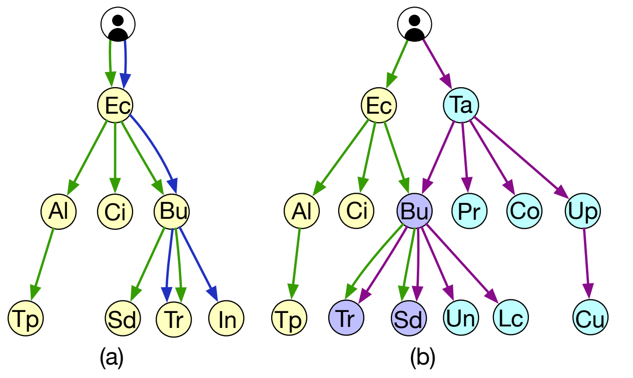

Figure 1: Microservice dependency graphs triggered by user requests collected from Alibaba traces. (a) Dependency graphs from the same online service can vary a lot. (b) Dependency graphs originated from two different entering microservices Ec and Ta share microservices (blue color).

(Sequence to Sequence) model [48], and BNN (Bayesian Neural Network) [50] based predictors. At the same time, Madu is able to achieve high training efficiency because the predictor can maintain good accuracy for a considerable period of time without retraining. In addition, we deployed the Madu system in a private cluster of 20 servers and evaluated its end-to-end performance using an open-source benchmark DeathStarBench [19] driven by Alibaba microservice traces. The experimental results show that compared with the existing reactive schemes, Madu can improve the cluster resource utilization by up to 1.7× and reduce the end-to-end latency of online services by 50%.

To summarize, we have made the following contributions in this paper:

- We design a new model to learn workload uncertainty for microservice workloads, which can handle dynamic dependencies between microservices and overcome underestimation at peak workloads;
- We implement a new proactive microservice autoscaler that can efficiently scale microservice containers with small overhead;
- We conduct real-world experiments by replaying traces from Alibaba production clusters to demonstrate the power of workload prediction.

2 BACKGROUND AND MOTIVATION

In this section, we present microservice workload characteristics and microservice resource usage metrics, which motivate us to design a new scaling system Madu.

2.1 Microservice Dependency Graphs

Under a microservice framework, the processing of each service request is represented by a dependency graph, as shown in Fig. 1. Each dependency graph is initiated by an entering microservice, such as Ec, and each service has a fixed entering microservice. In the graph, each node represents a distinct microservice with a unique name and each directed edge records a call between an upstream microservice and a downstream microservice.

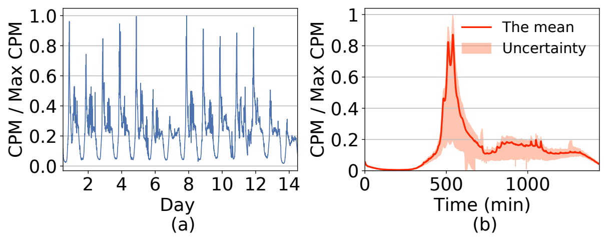

Figure 2: 14-days workloads of communication microservice from Alibaba clusters. (a) The workload of this microservice changes periodically over days. (b) Peak workload fluctuates more widely than other workloads. The solid line represents the average CPM produced at the same time on different days, and the shallow area represents the variance.

In production clusters, the microservice framework typically provides comprehensive business functions to satisfy users' various needs [32], and form highly dynamic dependencies between microservices accordingly. As depicted in Fig. 1(a), when serving two different user requests (represented by blue and green color, respectively) from the same online service Ec, their dependency graphs can vary in both structure and size. For example, the request in blue color contains only four microservices whereas the other request (in green color) includes seven microservices in different structure. Due to this highly dynamic feature, existing proactive autoscaling approaches cannot be practically used in production clusters because they all scale microservice containers based on static dependency graphs.

Furthermore, loosely-coupled microservices can be multiplexed between different services at runtime. As presented in Fig. 1(b), three microservices Bu, Tr, and, Sd are invoked by requests from both service Ec and service Ta. As a result, models built on dependency graphs of each individual service tend to misestimate resources for shared microservices. One way to improve performance is to combine dependency graphs from all online services and then optimize microservice performance based on the combined graph. However, this approach is not scalable because the combined graph could contain tens of thousands of microservices in a production cluster.

Takeaway. Dynamic dependencies and shared microservices bring great challenges for proactive microservice scaling. Existing graph-based solutions can not address these challenges. Madu turns to scale resources at the granularity

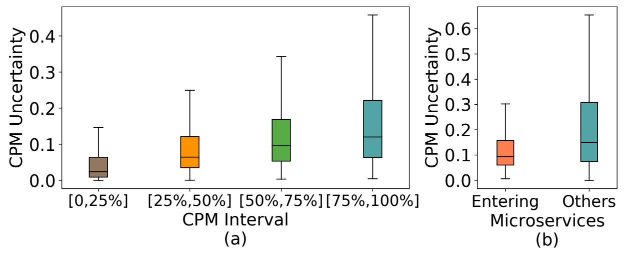

Figure 3: Quantification of microservice workload uncertainty, which is defined as the variance of workloads across different periods (relative to the mean workload). (a) The distribution of workload uncertainty under different CPM intervals. (b) The distribution of workload uncertainty for entering microservices and other non-entering microservices.

of a single microservice. By doing this, Madu can avoid modelling dynamic dependencies and does not require special effort to deal with shared microservices.

2.2 Microservice Workload Uncertainty

From the perspective of a single microservice, its performance depends heavily on the workload and the amount of resources allocated. Therefore, it is critical to model microservice workload for proactive scaling. Here, we define the workload as the number of calls per minute (CPM) a microservice needs to process.

In production clusters, microservice workload typically presents periodic patterns due to regular user activities. We show the communication microservice in Alibaba clusters in Fig. 2(a), whose workloads exhibit the same fluctuation pattern on weekdays. For example, the peak workload (the smallest workload) appears at the same time on each workday. However, microservice workloads also have strong data-dependent uncertainty. That is, the variance of workloads across periods is related to the mean workload. As revealed in Fig. 2(b), for the communication microservice, its peak workloads on different workdays have much higher variance than other workloads. To be more specific, we quantify workload uncertainty for each microservice in Alibaba clusters by computing the variance of workloads (relative to the average workload) across different periods over 30 days. As depicted in Fig. 3(a), the workload uncertainty grows with the average workload. At the peak workload, the highest variance can even reach as high as 0.5.

Microservice workload uncertainty is mainly caused by the dynamic dependency between microservices. As shown in Fig. 3(b), the uncertainty of non-entering microservices at peak workloads is much higher (2×) than that of entering microservices. Non-entering microservices are called by others at runtime. Depending on the dynamic topology of the dependency graph, their workloads can vary greatly across different periods.

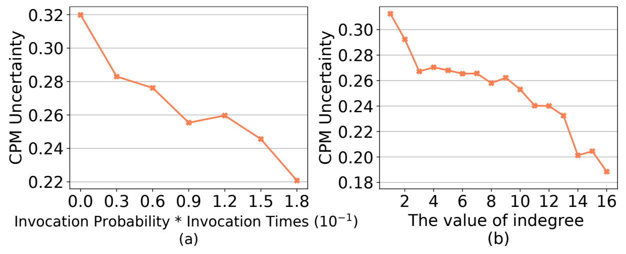

Figure 4: Relationship between workload uncertainty and dynamic dependency. (a) Workload uncertainty decreases as the product of invocation probability and invocation time increases. (b) Workload uncertainty decreases as the in-degree of microservices increases.

To further explore how dynamic dependencies affect the uncertainty of microservice workloads, we study microservice dependencies at runtime in terms of invocation time, invocation probability, and the in-degree of each microservice [31]. Invocation time measures how many times a microservice is called by its upstream microservices in a dependency graph, while invocation probability quantifies for each pair of microservices how likely a microservice is called by its upstream microservice. The smaller the invocation probability and the invocation time, the higher the dynamic dependency. In-degree counts the number of upstream microservices each microservice has. For example, Microservice Bu in Fig. 1 has an in-degree of two. The smaller the in-degree, the less chance a microservice tends to be called by its upstream microservices, resulting in higher dynamic dependency. As shown in Fig. 4(a), the workload uncertainty of microservices decreases as the invocation probability and invocation time increase. Similar observations can be found for the in-degree in Fig. 4(b).

Takeaway. The uncertainty of microservice workloads is data-dependent. Furthermore, microservices have non-uniform workload uncertainty depending on their specific dynamic dependencies. Madu needs to model such uncertainty in detail in order to accurately predict workload for efficient resource scaling.

2.3 Microservice Performance Metrics

When scaling containers for microservices, existing works mainly adopt application-level metrics such as microservice response time as a feedback signal [10, 41, 53]. However, in a microservice framework, the response time of a microservice does not reflect the actual overload as it may be severely impacted by its downstream microservices.

To demonstrate the ineffectiveness of microservice response time, we quantify its correlation with microservice

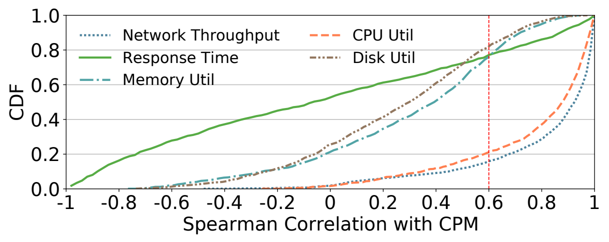

Figure 5: The distribution of Spearman correlation coefficient between microservice workload and OS-level resource utilization metrics (application-level response time) for microservices deployed in Alibaba clusters.

workload. Additionally, we examine how OS-level metrics, including CPU and memory utilization, disk utilization, and network throughput, relate to microservice workloads. As shown in Fig. 5, for all microservices in Alibaba clusters, OS-level metrics are more closely related to CPM than microservice response time. In particular, more than 80% (22%) of microservices have a strong correlation with Spearman coefficients [8, 54] greater than 0.6 between CPM and CPU utilization (memory utilization). Moreover, CPU and memory utilization perform similarly to network throughput and disk utilization, respectively.

Takeaway. CPU and memory utilization are much more strongly correlated with microservice workloads than microservice response time. Therefore, Madu turns to using these OS-level metrics to evaluate microservice performance for resource scaling.

3 SYSTEM OVERVIEW

We outline the workflow and main components of our designed auto-scaler, Madu. The overall architecture of the system is shown in Fig. 6. Details of all components are presented in § 4.

The auto-scaler runs as a separate entity in the microservice cluster. At the beginning of each minute, the auto-scaler queries the **Workload Predictor** (❶ in Fig. 6) to forecast the number of calls to be sent to each microservice in the future. The prediction model is trained based on historical workload traces from prior runs of all microservices. Using the predicted workload, the auto-scaler performs a **Utilization Analysis** (❸ in Fig. 6) to determine the initial number of containers that should be deployed in each interval so as to control CPU and memory utilization. Estimation of these utilization metrics for each microservice is initialized by the **Performance Profiling Model** (❷ in Fig. 6). When the microservice workload fluctuates widely, the initial scaling decision can easily result in a large scaling overhead and thus is not practically used. The auto-scaler sends these initial decisions to an **Autoscaling Optimization** (❹ in Fig. 6)

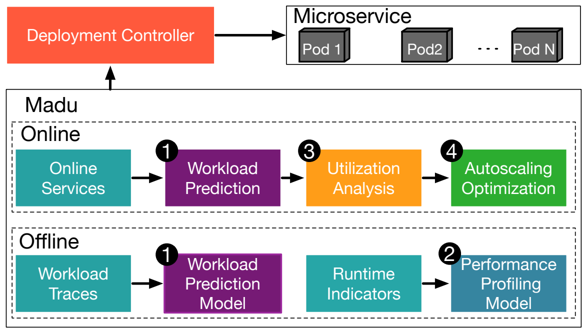

Figure 6: System architecture of Madu.

to determine the final number of containers to scale. The goal of this module is to avoid frequent scaling containers over long periods of time. These final decisions are then sent to the microservice cluster for actual execution.

4 DESIGN OF MADU

In this section, we present the design details of Madu. We first describe how Madu learns uncertainty to predict microservice workloads and uses linear regression to characterize microservice performance. Then, we explain how Madu scales containers based on workload prediction and performance characterization.

4.1 Microservice Workload Learning

The workload uncertainty of individual (shared) microservices learned from historical traces can indirectly reflect dynamic dependencies between related microservices. Modelling dependencies may help improve prediction accuracy, but it also incurs significant learning overhead since a microservice can have 100+ dependent microservices in production environment [31]. Therefore, Madu learns workload uncertainty without relying on microservice dependencies.

4.1.1 Madu's basic model. The key design behind Madu is the invention of the loss function (§ 4.1.3) and the integration of the stochastic attention mechanism (§ 4.1.4). Madu implements these designs on top of the basic *Seq2Seq* model, which is widely used in time-series prediction. It also worth noting that such designs can be combined with other deep neural network structures.

The model consists of an encoder and a decoder, as shown in Fig. 7. At each time t, the model takes a time-series sequence as its input and outputs a sequence as well. Specifically, the input of *Encoder* has a length of n and is represented by $X_t^E = [x_{t-n+1}, x_{t-n+2}, \cdots, x_t]$, and the output of the decoder has a length of m and is represented by $X_t^D = [x_{t+1}, x_{t+2}, \cdots, x_{t+m}]$. x_t denotes the feature vector at time t. The two parameters, n and m, are critical in Madu because they balance the tradeoff between resource scaling efficiency and prediction accuracy. On the one hand, the auto-scaler prefers a longer lookahead period to yield better scaling results. On the other hand, it is often difficult to predict long sequences with high accuracy.

4.1.2 Feature selection. Microservice workloads typically fluctuate periodically over days or weeks due to user activity. Therefore, we use time information as an additional feature of the workload to capture periodic patterns. Specifically, $x_t = \{C_t, t_m, t_h, t_w\}$ where C_t represents CPM, and t_m, t_h, t_w characterizes the exact minute, hour and week number at time t, respectively.

4.1.3 Uncertainty learning in Madu. For each microservice, the variance among all the CPM values produced at the same time from different periods (days or weeks) is viewed as uncertainty. As explained in § 2.2, this uncertainty is data-dependent. As such, the mapping from $\boldsymbol{X}_t^E$ to $\boldsymbol{X}_t^D$ should also carry data-dependent uncertainty in itself to model the uncertainty in the target $\boldsymbol{X}_t^D$. To achieve this, Madu formulates each target as $\boldsymbol{X}_t^D = f(\boldsymbol{X}_t^E) + \epsilon \cdot \sigma(\boldsymbol{X}_t^E)$. Note that Madu does not make any assumption on $\sigma(\boldsymbol{X}_t^E)$ and only assumes $\epsilon \sim \mathrm{N}(0, 1)$ follows a Gaussian distribution. As shown in the characterization study of Alibaba traces [32], Gaussian distribution matches the actual workload better than other distributions. Moreover, $f(\cdot)$ and $\sigma(\cdot)$ are two embedding functions with parameter $\boldsymbol{W}$, which are characterized by the *Seq2Seq* model. Madu estimates $\boldsymbol{W}$ using regression via maximizing the following likelihood function:

$$p(\boldsymbol{X}_t^D | \boldsymbol{X}_t^E, \boldsymbol{W}) = \exp\left(-\frac{(\boldsymbol{X}_t^D - f(\boldsymbol{X}_t^E))^2}{2\sigma^2(\boldsymbol{X}_t^E)}\right) \Big/ \sqrt{2\pi\sigma^2(\boldsymbol{X}_t^E)}. \tag{1}$$

As shown in Fig. 7, Madu trains the *Seq2Seq* network over parameter $\boldsymbol{W}$ to predict both the mean value and variance for each test point X_{test}^E, i.e.,

$$\left[\widehat{\mu}_{test}, \widehat{\sigma}_{test}\right] = \left[f(\mathrm{X}_{test}^E), \sigma(\mathrm{X}_{test}^E)\right], \tag{2}$$

where $\widehat{\mu}_{test}$ and $\widehat{\sigma}_{test}$ are a pair of values with the same dimension characterizing the estimated mean and variance respectively.

By designing the likelihood function in Eq. (1), the model encourages higher variance predicted for each $\boldsymbol{X}_t^E$ when the estimated mean $f(\boldsymbol{X}_t^E)$ deviates significantly from the true observation $\boldsymbol{X}_t^D$, which can well characterize data-dependent uncertainty.

4.1.4 Uncertainty-aware attention integrated to Madu. In the training phase, the output in *Decoder* and the input in *Encoder* are adjacent time-series data and therefore have similar uncertainty patterns. In addition to uncertainty learning in the output, Madu also introduces input-dependent uncertainty through a stochastic attention mechanism to further

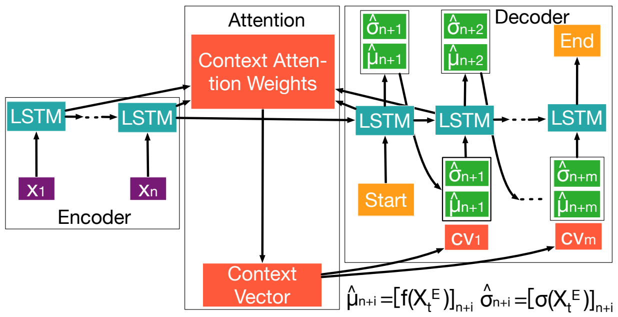

Figure 7: Design of workload learning model.

enhance prediction accuracy [24, 52]. Specifically, Madu assumes that the attention score $\boldsymbol{s}$ is sampled from a Gaussian distribution whose parameters are functions of input data $\boldsymbol{X}_t^E$ and network parameter $\boldsymbol{W}$, as shown in the following formula:

$$p(\mathbf{s} \mid \boldsymbol{X}_t^E, \boldsymbol{W}) = \mathrm{N}\left(\mu(\boldsymbol{X}_t^E, \boldsymbol{W}), \operatorname{diag}\left(\sigma^2(\boldsymbol{X}_t^E, \boldsymbol{W})\right)\right). \tag{3}$$

The key assumption about the attention score is that its generative probability $p(\mathbf{s} \mid \boldsymbol{X}_t^E, \boldsymbol{W})$ is conditional on $\boldsymbol{X}_t^E$ and shall capture the input-dependent uncertainty. When the input sequence contains a peak CPM value, the variance of the corresponding Gaussian distribution (per Eq. (3)) is expected to be high to produce an attention score with high uncertainty. With stochastic attention, the mean and uncertainty of the CPM prediction result shall be computed as:

$$\left[\widehat{\mu}_{t+1}, \widehat{\sigma}_{t+1}\right] = \left[f(\widehat{\mu}_t, \widehat{\sigma}_t, h_t, c_t, cv_t), \sigma(\widehat{\mu}_t, \widehat{\sigma}_t, h_t, c_t, cv_t)\right], \tag{4}$$

where h_t and c_t are the hidden state and the cell state of a corresponding LSTM unit in *Decoder*, respectively, and cv_t is a context vector that carries input-dependent uncertainty. By assigning different weights α_{tj} to each hidden state h_j^E of *Encoder*, i.e., $cv_t = \sum_{j=1}^n \alpha_{tj} h_j^E$, the context vector cv_t well preserves important information that helps to pay more attention to the burst traffic. In addition, the weight α_{tj} is computed as:

$$\alpha_{tj} = \frac{\exp\left(s_{tj}\right)}{\sum_{k=1}^n \exp\left(s_{tk}\right)}, \tag{5}$$

where attention score s_{tj} is given by:

$$s_{tj} = f(a_{tj}), \text{ and } a_{tj} = \tanh\left(W_a h_{t-1} + U_a h_j\right). \tag{6}$$

The variables W_a and U_a are parameters describing the attention mechanism. For deterministic attention, f is a linear function. In contrast, when implementing the stochastic attention mechanism, Madu generates s_{tj} by drawing random samples from the following Gaussian distribution $s_{tj} \sim \mathrm{N}(\mu_{tj}, \sigma_{tj})$ where $\mu_{tj} = W_{\mu j} a_{tj} + b_{\mu j}$, $\sigma_{tj} = W_{\sigma j} a_{tj} + b_{\sigma j}$.

Here, $W_{\mu j}$, $b_{\mu j}$, $W_{\sigma j}$ and $b_{\sigma j}$ are extra parameters for stochastic attention, which should be learned during the training process.

4.1.5 Distributed learning under MapReduce. When the number of microservices in the cluster is large, the model needs to be trained in parallel to enhance the scalability of the system. Madu adopts a machine learning framework, i.e., TensorFlow [7], for implementing deep learning algorithms, and deploys offline training on MaxCompute [36], a big data platform from Alibaba similar to Hadoop [23].

Map tasks fetch microservice traces from HDFS [46] to prepare the training dataset, while reduce tasks aggregate data from map tasks of each microservice and train the models. For the efficiency of online prediction, model parameters are written into object storage service (OSS), which is a low-latency key-value storage [5]. Online prediction itself is an online microservice that fetches model parameters from OSS to predict microservice workloads and sends the results back to the scaling modules, i.e., ❸ and ❹ in Fig. 6.

4.1.6 Online workload prediction. With uncertainty learning, the workload predictor needs to solve a key problem of efficiently combining the mean value $f(\boldsymbol{X}_t^E)$ and the variance $\sigma(\boldsymbol{X}_t^E)$ generated by the workload learning model.

Madu tackles this problem by maximizing the likelihood function in Eq. (1) under a fixed model parameter $\boldsymbol{W}$. Specifically, let $G(\sigma(\boldsymbol{X}_t^E)) = \ln\left(\prod_{t=1}^{T} p(\boldsymbol{X}_t^D | \boldsymbol{X}_t^E, \boldsymbol{W})\right)$ where p is given by Eq. (1), the derivative of $G(\sigma(\boldsymbol{X}_t^E))$ with respect to $\sigma^2(\boldsymbol{X}_t^E)$ is given by:

$$\frac{\partial G}{\partial \sigma^2(\boldsymbol{X}_t^E)} = -\frac{(\boldsymbol{X}_t^D - f(\boldsymbol{X}_t^E))^2}{\sigma^4(\boldsymbol{X}_t^E)} + \frac{1}{\sigma^2(\boldsymbol{X}_t^E)}. \tag{7}$$

To maximize the likelihood function, the optimum is attained when $\sigma^2(\boldsymbol{X}_t^E) = (\boldsymbol{X}_t^D - f(\boldsymbol{X}_t^E))^2$. Based on this result, Madu obtains its final prediction for each input $\mathbf{X}_{test}^E$ as follows:

$$\widehat{\mathrm{Y}}_{test} = \widehat{\mu} + \widehat{\sigma} = f(\mathbf{X}_{test}^E) + \sigma(\mathbf{X}_{test}^E). \tag{8}$$

One can also maximize the likelihood function with respect to X_t^D. However, taking derivative on X_t^D will lead to a result that completely ignore the effect of uncertainty, which violates the observations in this paper.

4.2 Microservice Performance Profiling

Madu adopts CPU and memory utilization as performance metrics for scaling microservice containers. Let $g_i^{CPU}(\cdot)$ and $g_i^{Mem}(\cdot)$ denote the CPU and memory utilization estimation of a deployed container under a given workload for each microservice i.

As depicted in Fig. 8, both traces from Alibaba clusters and real benchmarks show that the average CPU and memory utilization of a running container within a minute grows almost

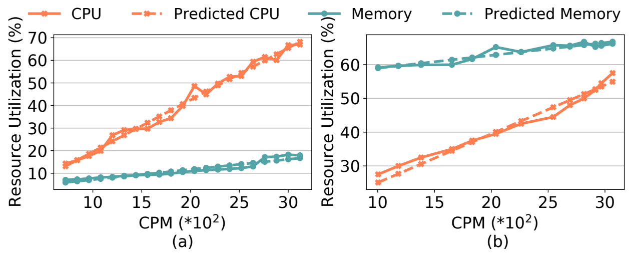

Figure 8: CPU and memory utilization of microservice containers grows linearly with microservice workloads. (a) Microservices from Alibaba traces [2]. (b) Microservices from DeathStarBench [19].

linearly in CPM. As such, Madu adopts a linear regression model to profile $g_i(\cdot)$ for memory and CPU utilization,

$$g_i^{CPU}(x) = a_i^{CPU} \cdot x + b_i^{CPU}, \tag{9}$$

$$g_i^{Mem}(x) = a_i^{Mem} \cdot x + b_i^{Mem}, \tag{10}$$

where x is per-container CPM of Microservice i, a_i^{CPU} (a_i^{Mem}) and b_i^{CPU} (b_i^{Mem}) are the slope and intercepts of the regression equations.

We adopt traces generated from DeathStarBench containing nearly 90 microservices, as well as Alibaba clusters with more than 1000 microservices to validate the efficiency of the linear regression model. The result shows that the estimation accuracy under these two traces can be as high as 90% and 89%, respectively.

4.3 Online Container Scaling

4.3.1 Utilization analysis. Based on the CPU and memory utilization profile and predicted workloads, the autoscaler scales containers for each microservice so that the final CPU and memory usage of each container does not surge. Specifically, Madu formulates a resource minimization problem to configure the number of containers $c_i(t)$ for each microservice i at time t as following:

$$\begin{aligned} \min_{c_i(t)\in\mathcal{N}} \quad & c_i(t) \\ \text{s.t.} \quad & g_i^{CPU}\left(L_i(t)/c_i(t)\right) \le T_i^{CPU}, \\ & g_i^{Mem}\left(L_i(t)/c_i(t)\right) \le T_i^{Mem}, \end{aligned} \tag{11}$$

where $L_i(t)$ is the estimated workload for microservice i at time t, T_i^{CPU} and T_i^{Mem} are predefined CPU and memory thresholds, respectively. The solution to this optimization problem is given by:

$$c_i(t) = \max\left\{\left\lceil \frac{a_i^{CPU}}{T_i^{CPU} - b_i^{CPU}} \cdot L_i(t) \right\rceil, \left\lceil \frac{a_i^{Mem}}{T_i^{Mem} - b_i^{Mem}} \cdot L_i(t) \right\rceil\right\}. \tag{12}$$

4.3.2 Autoscaling optimization. Simply following the above result may lead to frequent scaling out and scaling in, especially when the workload fluctuates widely between successive time intervals. Therefore, Madu tries to avoid scaling a large number of containers in each interval (one minute long) by formulating an additional optimization problem:

$$\min_{\boldsymbol{x}_i} \sum_{k=1}^{m} \left(x_i(t+k-1) - x_i(t+k)\right)^2 \tag{13}$$

$$\text{s.t., } c_i(t+k) \le x_i(t+k) \le (1+\rho) \cdot c_i(t+k). \tag{14}$$

In the t-th time interval, the auto-scaler minimizes the total number of containers to be scaled in the next m control intervals, where m is the length of the lookahead period under Madu. We take square in the objective to make the scaling smoother when the cluster is overloaded. The constraint (per Eq. (14)) states that, the number of deployed containers $x_i(t)$ should be no less than $c_i(t)$ (characterized by Eq. (12)) to guarantee microservice performance and in the meanwhile, $x_i(t)$ is not far away from $c_i(t)$ to ensure high utilization. ρ is a parameter that balances this performance and utilization trade-off.

A greedy solution for this optimization problem that picks $x_i(t+1) = \max\left\{\min\left\{x_i(t), (1+\rho)c_i(t+1)\right\}, c_i(t)\right\}$ only minimizes the scaling overhead within two successive time intervals and can easily lead to suboptimal solutions over a long control window. Since this optimization problem is convex, Madu adopts CVX solver [11] to obtain the optimal solution for scaling in the next m intervals, which can yield low complexity. For instance, when $m = 30$, the optimization problem can be solved within only 10ms on Intel Xeon E5-2630 CPU. Moreover, as described in § 4.1, a long lookahead period, i.e., a large m will negatively affect the prediction accuracy of Madu's predictor, resulting in poor scaling results. In contrast, a small m causes the auto-scaler to make almost greedy decisions. We configure an appropriate lookahead period for Madu by benchmarking the highly fluctuated workload, while varying the value of m and selecting the point with the best scaling efficiency and end-to-end performance. We discuss the choices for real benchmarks in experiments in § 5.2.2.

4.4 System Implementation

We implemented Madu on top of Kubernetes [28]. A Kubernetes cluster is deployed with two open-source tracing systems: Jaeger [3] and Prometheus [6]. Jaeger is a system to collect application-level metrics, including all calls send to each microservice and service response time. In the implementation, we set the sampling ratio of Jaeger to 10% to control the data collection overhead. Prometheus collects OS-level metrics including CPU and memory utilization for each running container of a microservice.

Madu's workload predictor fetches workload traces from Prometheus and Jaeger, and trains the prediction model via TensorFlow under the MapReduce framework (§ 4.1.5). To reduce training overhead, the predictor adopts incremental training and updates model parameters once a week. At the beginning of each m intervals, the auto-scaler determines the number of containers to scale in each interval. In addition, the auto-scaler relies on the Kubernetes default deployment controller to perform actual scaling and prepare the running environment in advance for the deployment of newly scheduled containers.

5 PERFORMANCE EVALUATION

In this section, we first evaluate the performance of Madu's workload predictor using Alibaba cluster traces. We then evaluate the scaling performance of Madu using an open-source microservice benchmark DeathStarBench [19].

5.1 Evaluation of Workload Predictor

We conducted large-scale experiments in Alibaba clusters using more than 1000 microservices to demonstrate the efficiency of Madu's workload predictor. We collected 30-day traces to train the model and conducted validation using 10-day traces. We present the evaluation results in several aspects below.

5.1.1 Prediction accuracy. We compare Madu's workload predictor (using n = 60 and m = 5) with other baselines, as described in Table 1. The prediction accuracy is defined as:

$$\textit{prediction accuracy} = \frac{1}{N}\sum_{test}\left(1 - \frac{|\mathbf{X}^D_{test} - \widehat{\mathbf{Y}}_{test}|}{\mathbf{X}^D_{test}}\right), \tag{15}$$

where N is the number of testing samples. Table 2 reports the average accuracy of all microservices under different predictors. To be more comprehensive, we divide the CPM values of each microservice into three groups: below 50th percentile, from 50th to 95th percentile, and above 95th percentile. We then quantify the prediction accuracy of each group and take the average across all microservices. It is worth noting that, for small CPMs in the first group, the prediction accuracy of ARIMA and BNN is below 75%, whereas the accuracy achieved by Madu exceeds 90%. When compared to DUBNN, Madu can improve the accuracy by 26%. For the third group consisting of large workloads, Madu can achieve 1% higher accuracy than DUBNN. In this sense, Madu is demonstrated to be robust to deal with burst traffic. Moreover, when referring to the average performance, Madu could outperform other baselines by 13%.

To examine how likely an individual prediction is accurate, we quantify the percentage of estimations that are accurate enough in each group, i.e., within $(1 \pm 10\%) \times$ ground truth.

Table 1: Baseline workload predictors

Model	Description
ARIMA	Autoregressive integrated moving average model [15].
Seq2Seq	Sequence-to-Sequence model with a deterministic output [48].
BNN	Conventional BNN model whose output is a range instead of a constant [50].
DUBNN	Bayesian Neural Network model with Data-dependent Uncertainty, it uses dropout [47] to approximate BNN [27].

Table 2: Prediction accuracy for all microservices under different CPM percentiles (n=60, m=5)

Percentile	ARIMA	Seq2Seq	BNN	DUBNN	Madu
[0,50%]	72.1	83.6	73.3	74.4	91.1
[50%,95%]	86.1	87.1	89.4	88.7	93.8
[95%,100%]	88.8	89.7	89.2	90.8	91.5
Avg	79.3	87.1	81.3	81.6	92.3

As illustrated in Fig. 9(a), around 80% of estimations are accurate for the second and third groups of CPMs. In contrast, in the first group, the percentage of accurate predictions under Madu drops to 70%. Madu can easily make overestimations due to the adoption of uncertainty learning. Nevertheless, on average, Madu can increase the number of accurate estimations by more than 40% compared to the other four predictors. In the first two groups, both BNN and DUBNN can cause 60% of estimations to violate the ground truth by more than 10%, implying BNN-related models can lead to severe performance degradation in production clusters.

To explore how far the overestimation can be from the ground truth, we quantify the overestimation accuracy for each predictor, which is defined as:

$$overestimation\ accuracy = \frac{\sum_{\widehat{Y}_{test}>X^D_{test}} X^D_{test}/\widehat{Y}_{test}}{\sum_{\widehat{Y}_{test}>X^D_{test}} 1}. \quad (16)$$

Fig. 9(b) shows that for small CPMs, the overestimation accuracy of baseline predictors is 70%, much lower than Madu, which is 92%.

5.1.2 Case study. To highlight the effectiveness of Madu, we take a representative communication microservice as an example. As can be observed from Fig. 10(a), ARIMA is prone to bad predictions at inflection points. The *Seq2Seq* model is able to achieve high prediction accuracy for most CPMs, except for the peaks in the first four days, which are often underestimated, as shown in Fig. 10(b). Without uncertainty learning, traditional deterministic models have difficulty predicting

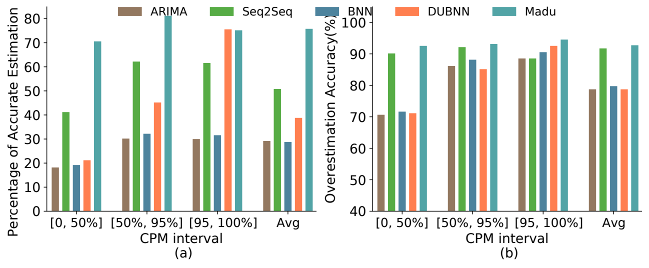

Figure 9: Prediction performance under different CPM percentiles. (a) The percentage of accurate predictions. (b) Overestimation accuracy.

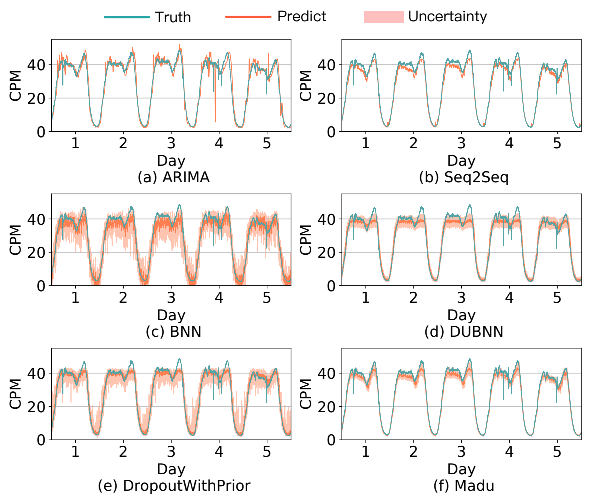

Figure 10: Predictions for communication microservice.

bursty workloads. When referring to BNN in Fig. 10(c), even for a low CPM, the 95th percentile of the estimated values is much higher than the median, which substantially limits the applicability of BNN in practice. In addition, training a BNN network requires one to evaluate the posterior distribution of network parameters, which is computationally intractable in general. To overcome this limitation, DUBNN integrates into the network a dropout mechanism as an approximation of BNN. Meanwhile, DUBNN also predicts the data-dependent uncertainty. Fig. 10(d) shows that DUBNN does not perform much better than *Seq2Seq* since its estimation of the peak CPM is similar to that achieved by the latter. This is caused by the inappropriate assumption on the distribution of the dropout probability. Specifically, there is a large gap between $p(\theta|D)$ (true distribution) and $q(\theta)$ (priori assumption) where D is the training dataset and θ is the parameter characterizing the distribution. To validate this argument, we illustrate in Fig. 10(e) the performance of dropout relying on a priori assumption. The results show the dropout scheme performs worse than the basic *Seq2Seq*

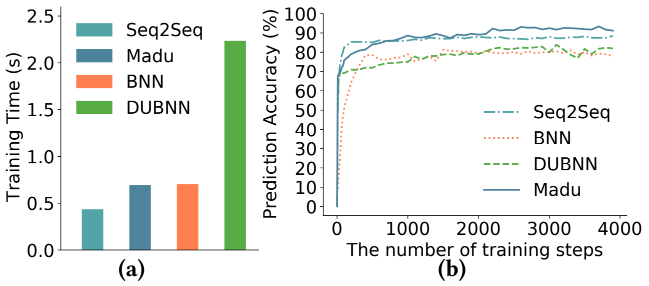

Figure 11: Training performance under different predictors. (a) Training overhead on NVIDIA Tesla v100. (b) Prediction accuracy under different training steps.

model, indicating that an imperfect assumption can easily degrade the prediction performance. Instead, Madu learns data uncertainty directly without relying on BNN framework, thereby resulting in much better prediction results.

5.1.3 Training overhead v.s. efficiency. In this part, we investigate the overhead of uncertainty learning. As shown in Fig. 11(a), we quantify the average training time for a mini-batch of all microservices on NVIDIA Tesla v100 under different predictors. The results show that among all these four predictors *Seq2Seq* has the shortest training time. By using uncertainty learning and stochastic attention, Madu increases training time by only 60% compared to *Seq2Seq* (0.69s vs. 0.43s), and performs similarly to the basic BNN model (i.e., 0.69s vs. 0.71s). In contrast, DUBNN takes three times longer to train a mini-batch than all other predictors. This large overhead is caused by the dropout mechanism, which requires sampling the entire network to determine whether a neuron in each LSTM unit should be activated. Furthermore, ARIMA should be updated at runtime and retraining each microservice takes more than 10 seconds using ARIMA on Intel Xeon E5-2630 CPU.

To evaluate the training efficiency, we characterize how prediction accuracy increases with the number of training steps under all predictors, as shown in Fig. 11(b). In the first 100 training steps, the prediction accuracy under *Seq2Seq* is the highest due to its simple structure, which is 6.7%, 27.5%, and 8.15% higher than Madu, BNN, and DUBNN, respectively. After that, the prediction accuracy under *Seq2Seq* remains stable, while it continues to improve under other predictors. Specifically, by increasing the number of training steps from 100 to 3000, the prediction accuracy under Madu is further improved by 16.6%. Compared with these predictors, ARIMA directly updates its model parameters and does not use iterative methods, so it achieves the highest training efficiency.

5.1.4 Accuracy under different prediction lengths. We proceed to evaluate the prediction accuracy of all microservices under different combinations of n (the length of input) and m (the length of output). As shown in Table 3, the prediction accuracy will improve significantly with the increase of n.

Table 3: The prediction accuracy of Madu under different combinations of n and m

	n = 30	n = 60	n = 120
m=5	87.0	92.3	92.8
m=30	81.6	85.5	88.4
m=60	78.5	83.5	87.6

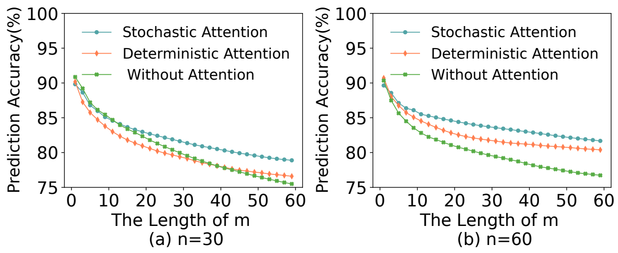

Figure 12: Prediction results under different n and m with or without attention mechanism. (a) n = 30. (b) n = 60.

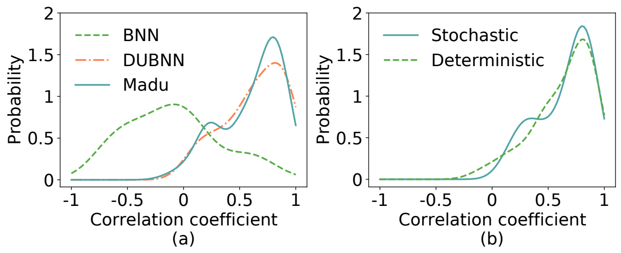

Figure 13: The probability density function (PDF) of the correlation coefficient between the actual variance and the predicted uncertainty under different predictors with and without the use of attention. (a) Without attention. (b) With different attention mechanisms.

To investigate the benefit of attention, we quantify the prediction accuracy of different attention mechanisms for all microservices, as shown in Fig. 12. When $n = 60$ and $m = 5$, stochastic attention brings an improvement of 4.4% against predictors without attention. As a comparison, for the configuration where $n = 60$ and $m = 60$, the prediction accuracy increases more than 12% with the use of stochastic attention. This result implies that stochastic attention is very effective for improving the prediction accuracy of long sequences.

5.1.5 Why uncertainty learning? To figure out why uncertainty learning significantly improves prediction accuracy, we quantify the estimated uncertainty under all predictors and in addition, compute the correlation coefficient between this learned uncertainty and the actual variance of CPM of all microservices. The actual variance is obtained by collecting all samples at the same time across all periods. Fig. 13(a)

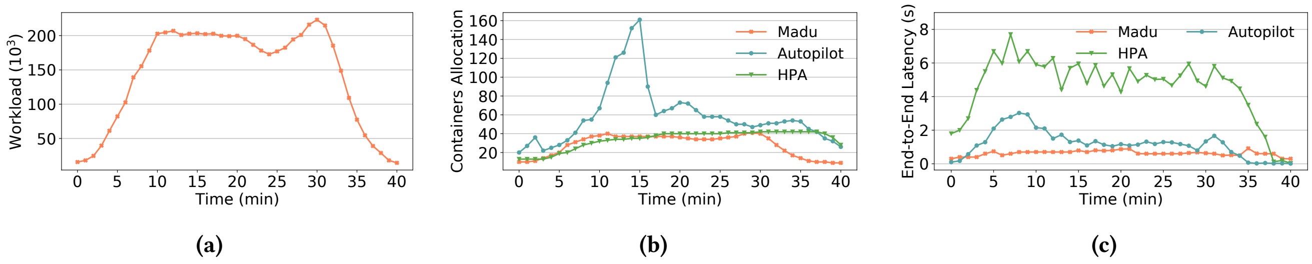

Figure 14: Comparison between reactive and proactive schemes using Media service. (a) Smoothly-fluctuated workload from Alibaba traces. (b) Resource allocation overtime. (c) 95th percentile end-to-end latency of online services.

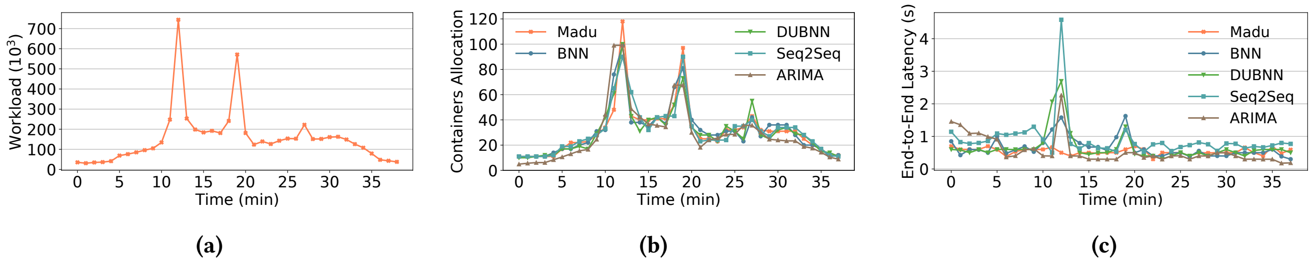

Figure 15: Comparison between different predictor-based proactive scaling schemes. (a) Heavily-fluctuated workload from Alibaba traces. (b) Resource allocation in runtime. (c) 95th percentile end-to-end latency of online services.

shows that BNN does not learn the data uncertainty well because it only takes into account the model uncertainty by assuming that the network parameters follow a random distribution. As a comparison, most of the uncertainty learned from Madu and DUBNN is strongly correlated with the actual CPM variance because of their predictions of data-dependent uncertainty. Moreover, Madu also outperforms DUBNN as its average correlation coefficient increases by 4% compared to the latter.

Fig. 13(b) depicts the learned uncertainty under both deterministic attention and uncertainty-aware (stochastic) attention. The results also show that stochastic attention helps to estimate uncertainty more accurately (the average correlation coefficient under stochastic attention is 2% larger than that under deterministic attention).

5.2 Results of Container Scaling

We conducted extensive experiments to evaluate the end-to-end performance of Madu's auto-scaler including the resource usage and service latency. We introduce the experiment setup below.

Benchmarks. We use a widely-adopted and open-sourced microservice benchmark, DeathStarBench [19], consisting of three applications, Social Network, Media Service, and Hotel Reservation. These applications include 3, 1, and 2 different online services, and, contain 36, 38, and 15 unique microservice respectively.

Cluster Setup. We deployed the auto-scaler in a local private cluster, consisting of 20 two-socket physical nodes. Each node is configured with 32 CPU cores and 64 GB RAM. All CPU cores are Intel Xeon E5-2630.

Workload. Our experiments are driven by workloads from Alibaba cluster traces [2]. Since Alibaba clusters have deployed thousands of online services, far more than DeathStarBench, we selected only two representative workload traces. In particular, we chose one workload that fluctuates smoothly and another that fluctuates widely. In addition, to accommodate our cluster size, we also scaled both workloads to avoid multiple requests being dropped.

Baseline Schemes. We compare our auto-scaler ($m = 5$, $\rho = 0.2$ as discussed in § 4.3) against Google Autopilot [44] and Kubernetes default horizon pod auto-scaler (HPA) [4]. Autopilot is a reactive scaler that uses a moving window to collect resource usage statistics in a most recent period, and scales resources based on these statistics, such as the average CPU and memory utilization. The autoscaling method under Kubernetes is similar to Autopilot, except that the latter adopts a more accurate estimator for resource utilization.

5.2.1 Improvement from proactive scaling. In this part, we investigate the improvement brought by proactive scaling. In particular, we compare the run-time metrics across Autopilot, HPA, and Madu, as shown in Fig. 14. In this comparison, we run workloads with small fluctuations over 40 minutes under Media service (Fig. 14(a)). The predefined thresholds

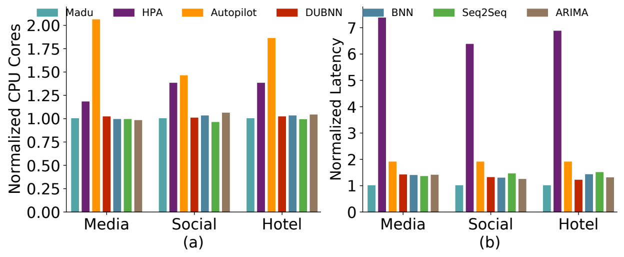

Figure 16: Comparison between different scalers in terms of resource usage and end-to-end latency.

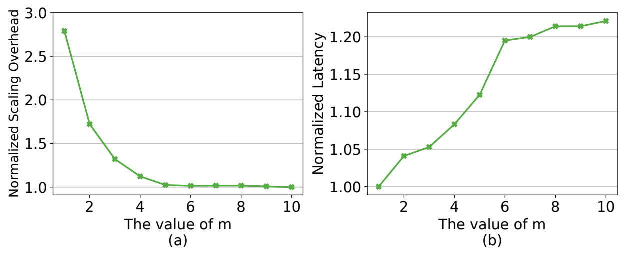

Figure 17: The trade-off between scaling overhead and microservice performance under different m.

of CPU and memory utilization are set to 50% and 60% for all microservices, respectively. As shown in Fig. 14(b), Madu can significantly reduce the number of allocated containers compared to Autopilot. On average, it can save resource usage by up to 1.7×. When workload increases slowly, Autopilot can actually learn this trend based on historical CPU and memory utilization and allocates resources aggressively in case of further increase of workload in the future. Therefore, Autopilot can easily lead to a huge waste of resources. At the 15th minute, the number of containers allocated under Autopilot is four times as that under Madu. However, such allocations are still too late to meet growing resource demands because it takes time to deploy containers in a cluster. Furthermore, the resource scaling for bottleneck microservice is even much later than other microservices. For those microservices, they need to handle call requests from multiple upstream microservices. Therefore, reactive scaling can greatly degrade the response performance of bottleneck microservices, delaying the end-to-end service response time. As shown in Fig. 14(c), the maximum 95th percentile service latency under Autopilot is three times larger than that under Madu.

HPA allocates resources more conservatively, as it deploys additional containers only when the condition is met, i.e., the average CPU and memory utilization of running containers over the past period (30 seconds in default) have exceeded 50% and 60%, respectively. Therefore, HPA consumes only 25% more resources compared to Madu. However, such a short period limits the reaction time of HPA as well as its efficiency for handling the peak demand. As shown in Fig. 14(c), the end-to-end latency of HPA and autopilot is 6.37× higher than that of Madu on average. When referring to latency performance at the peak demand, HPA performs even worse, namely a 10× degradation. In addition, both HPA and Autopilot tend to delay scaling of containers when CPM starts to drop at the 30th minute, while Madu could reclaim resources on-demand and result in stable end-to-end latency. These results demonstrate that active scaling using workload prediction can achieve higher resource utilization and, more importantly, yield better service performance compared to reactive approaches.

Moreover, to figure out how accurate workload prediction can help to improve performance of proactive scaling, we conducted another experiment to evaluate auto-scalers using different predictors under highly fluctuated workloads, as shown in Fig. 15(a). Fig. 15(b) depicts the number of containers allocated under all scalers. It shows that all these auto-scalers can act promptly when the workload suddenly fluctuates. Madu uses 3% more resources than the other four scalers due to underestimating peak demands. Nonetheless, this small additional resource usage helps provide better service performance for Madu. As illustrated in Fig. 15(c), Madu manages to reduce end-to-end latency by 35% on overage and 5.9× at peak demands, compared to the other four scalers. This result shows again that accurate uncertainty prediction is critical to ensuring service SLA, especially when workload fluctuates widely.

For a more comprehensive study, we compare all auto-scalers among three applications. As shown in Fig. 16(a), Madu can reduce resource usage by 40% and 1×, compared to HPA and Autopilot, respectively. Moreover, different proactive auto-scalers result in almost the same resource usage. Fig. 16(b) shows that Madu reduces the average end-to-end latency by 31%, 43%, 36%, and 30% compared with ARIMA-, Seq2Seq-, BNN-, and DUBNN-based auto-scalers, respectively. When compared to Autopilot and HPA, Madu can improve latency performance by 5.87× and 90%.

5.2.2 *The trade-off between performance and scalability.*

When deploying Madu in production environments, it is critical to balance the tradeoff between system scalability and service performance. In this experiment, we tuned the length of the lookahead period (i.e., the value of m) to investigate such a tradeoff. As shown in Fig. 17(a), increasing m from 1 to 5 can greatly reduce the number of containers that need to be scaled in each interval, i.e., the scaling overhead is reduced by 2.3×. However, larger m also leads to higher prediction errors, which can greatly degrade service performance. The results in Fig. 17(b) show that, when $m = 5$ ($m = 10$), the worst end-to-end latency is 10% (23%) higher than when $m = 1$. In our design, we set m to 5 to achieve the best tradeoff between the scaling overhead and the latency performance, others may choose different values based on the importance of these two metrics.

5.2.3 Overhead of auto-scaler. Our auto-scaler Madu can find the best configuration for scaling with little overhead. For an application consisting of 80 microservices, it takes less than 10ms to compute the optimal number of containers to scale for all microservices.

6 DISCUSSION

In this section, we discuss how Madu adapts to environmental changes and whether Madu can leverage a unified model to achieve accurate workload predictions.

6.1 Adaptivity of Madu

Madu achieves proactive scaling by predicting microservice workloads in advance and estimating resource utilization. A key question that arises when deploying Madu in a production cluster is how Madu adapts to different environmental changes. Here, we discuss three representative scenarios, workload pattern changes, microservice evolves, and rare events.

6.1.1 Workload pattern changes. When users' regular activities change or new (old) microservices are added (removed), the fluctuation pattern of microservice workloads shall change accordingly. In response to this change, Madu employs incremental training for new traces once a week to maintain high prediction accuracy and low overhead. This retraining interval is consistent with the normal period of software development in Alibaba. We validated incremental training results for over 1000 microservices using traces generated in the most recent week, showing that the retrained model improves prediction accuracy by only 1.5% on average. This result demonstrates Madu's workload learning model can adapt to workload changes over a considerable period of time.

6.1.2 Microservice evolves. The continuous development of microservices often leads to changes in resource utilization for the same workload. Madu needs to update the performance profiling model as the microservice evolves. However, the profiling cost under Madu is negligible because it only uses linear regression. As such, evolving microservice does not incur extra overhead on Madu.

6.1.3 Handling rare events. All proactive auto-scalers designed for predicting periodic workloads perform unsatisfactorily for rare events. Production clusters usually reserve enough resources in advance to handle rare peak workloads, for example, Alibaba prepares enough resources for the annual shopping Festival.

6.2 Comparison Across Different Models

Madu implements an individual model for each microservice for high accuracy of workload prediction and high scheduling scalability. In this part, we will discuss the possibility of using other unified end-to-end models.

6.2.1 Individual model vs. unified model. Madu builds an individual workload learning model independently for each microservice instead of using a unified model for all microservices. There are two key reasons behind this design. First, microservices often provide many different business functions, so their workloads have non-uniform fluctuation patterns. Second, microservices have different dynamic dependencies, resulting in varying degrees of workload uncertainty. Therefore, building a separate model for each microservice can achieve higher prediction accuracy, which is critical for proactive scaling.

6.2.2 Separate model vs. end-to-end model. Instead of implementing a single end-to-end model, Madu combines separate models (i.e. workload learning model and container prediction model) to scale containers for microservices. The reason is that end-to-end model needs to evaluate a large number of resource configurations to find an optimal allocation [55], which is not scalable in production clusters. Without end-to-end training, Madu can still reduce resource usage by 1.7× due to its accurate workload prediction.

7 RELATED WORK

Workload prediction has been an active research topic in recent years [13, 33, 34, 38, 44, 45]. These works adopt different approaches to predict time series sequence including resource usage and workloads of online services for the purpose of resource auto-scaling. In particular, CloudScale applies Fast Fourier Transform (FFT) to identify repeating patterns of resource usage and tune the amount of resources allocated to virtual machines accordingly [45]. Similarly, AGILE employs wavelet transform to predict the CPU demand of web services [38]. In [13], the authors combined ARIMA and exponential smoothing to predict the long-term resource usage for virtual machine resource reclamation. Ma *et al.* adopted an ensemble method that utilizes RNN, linear regression, and kernel regression to predict the arrival rate of queries in database systems [33]. Google Autopilot performs exponential smoothing with a sliding window over historic resource usage to predict the actual resource demand of each task [44]. However, none of these works applied data-dependent uncertainty in predicting the variance of highly-fluctuated workload.

Existing works leverage Bayesian models for learning uncertainty [14, 24, 27, 51]. In [27], the authors built a Bayesian deep learning framework combining data uncertainty (the noise inherent in given training data) together with model uncertainty (i.e., the network parameters are sampled from a random distribution) to deal with per-pixel semantic segmentation. Researchers also applied data uncertainty prediction on top of Bayesian models for handling NLP tasks [51] and more general computer vision applications [14, 24].

Another research direction proposes to study the distributional uncertainty caused by the mismatch between the test and training data distributions [29, 35, 39]. However, these methods are based on the Bayesian frameworks, which usually take a long time to train. As a comparison, Madu adopts a deterministic model without relying on BNN models for handling data-dependent uncertainty of microservice workloads and is more computationally efficient.

Most recently, researchers began to design auto-scalers for microservice-based applications [9, 18, 25, 26, 37, 41, 43, 55, 56]. These works mainly focus on optimizing the violation probability of end-to-end SLA requirements using reactive or proactive solutions. The proposed auto-scalers took as input a static microservice dependency graph, which would not change over time. As a result, these auto-scalers can not be applied to practical scenarios where microservice call graphs are dynamic in runtime. In contrast, our designed Madu focuses on each individual microservice instead of a whole dependency graph.

8 CONCLUDING REMARKS

In this paper, we have designed a proactive auto-scaler in a microservice framework by predicting workloads and characterizing performance for each individual microservice. Our auto-scaler can achieve much better service performance than existing solutions. The key novelty behind the design is to leverage uncertainty learning that can handle dynamic microservice dependencies and adopt OS-level metrics for more efficient resource scaling. One limitation of our design is that it can not achieve optimal end-to-end resource efficiency given the service SLA requirements. Exploring this problem in the context of dynamic microservice dependencies and microservice multiplexing will be our future work.

9 ACKNOWLEDGEMENTS

We sincerely thank the anonymous SoCC'22 reviewers and our shepherd, Yongjoo Park, for their insightful suggestions. This work is supported in part by Key-Area Research and Development Program of Guangdong Province under Grant No. 2020B010164003, the National Natural Science Foundation of China (No. 62072451), the Start-up Research Grant of University of Macau (SRG2021-00004-FST), and Alibaba Group through Alibaba Innovative Research Program.

REFERENCES

[1] 2022. Alibaba Cloud. https://www.alibabacloud.com/.
[2] 2022. Alibaba Microservice Traces. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022.
[3] 2022. Jaeger: Open source, end-to-end distributed tracing. https://jaegertracing.io/.
[4] 2022. Kubernetes Horizon Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[5] 2022. Object Storage Service. https://www.alibabacloud.com/product/oss.
[6] 2022. The Prometheus monitoring system and time series database. https://github.com/prometheus/prometheus/.
[7] Martín Abadi, Paul Barham, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Processing of OSDI*.
[8] Haldun Akoglu. 2018. User's guide to correlation coefficients. *Turkish journal of emergency medicine* (2018).
[9] Ataollah Fatahi Baarzi and George Kesidis. 2021. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*.
[10] Luciano Baresi, Sam Guinea, Alberto Leva, and Giovanni Quattrocchi. 2016. A Discrete-Time Feedback Controller for Containerized Cloud Applications. In *Proceedings of FSE*.
[11] S. Becker, E. J. Candès, and M. Grant. 2011. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical Programming Computation* (2011).
[12] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proceedings of SoCC*.
[13] Marcus Carvalho, Walfredo Cirne, et al. 2014. Long-term SLOs for reclaimed cloud computing resources. In *Proceedings of SoCC*.
[14] Jie Chang, Zhonghao Lan, et al. 2020. Data Uncertainty Learning in Face Recognition. In *Proceedings of CVPR*.
[15] Chris Chatfield. 2003. *The analysis of time series: an introduction*. Chapman and hall/CRC.
[16] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. 2022. DeepRest: Deep Resource Estimation for Interactive Microservices. In *Proceedings of EuroSys*.
[17] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of ASPLOS*.
[18] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical & Scalable ML-Driven Performance Debugging in Microservices. In *Proceedings of ASPLOS*.
[19] Yu Gan, Yanqi Zhang, et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of ASPLOS*.
[20] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of ASPLOS*.
[21] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-Driven Autoscaling for Microservices. In *Proceedings of ICDCS*.
[22] GoogleCloud. 2022. https://cloud.google.com/.
[23] Hadoop. 2022. https://hadoop.apache.org/.
[24] Jay Heo, Hae Beom Lee, et al. 2018. Uncertainty-aware attention for reliable interpretation and prediction. In *Proceedings of NeurIPS*.

[25] Md Rajib Hossen, Mohammad A Islam, and Kishwar Ahmed. 2022. Practical Efficient Microservice Autoscaling with QoS Assurance. In *Proceedings of HPDC*.

[26] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, and Jason Mars. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of EuroSys*.

[27] Alex Kendall and Yarin Gal. 2017. What uncertainties do we need in bayesian deep learning for computer vision?. In *Proceedings of NeurIPS*.

[28] Kubernetes. [n.d.]. https://kubernetes.io..

[29] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Proceedings of NeurIPS*.

[30] Qixiao Liu and Zhibin Yu. 2018. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba trace. In *Proceedings of SoCC*.

[31] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of SoCC*.

[32] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Cheng-Zhong Xu. 2022. An In-depth Study of Microservice Call Graph and Runtime Performance. *IEEE Transactions on Parallel and Distributed Systems* (2022).

[33] Lin Ma, Dana Van Aken, et al. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of SIGMOD*.

[34] Ashraf Mahgoub, Alexander Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2020. OPTIMUSCLOUD: Heterogeneous Configuration Optimization for Distributed Databases in the Cloud. In *Proceedings of ATC*.

[35] Andrey Malinin and Mark Gales. 2018. Predictive uncertainty estimation via prior networks. In *Proceedings of NeurIPS*.

[36] MaxCompute. [n.d.]. https://www.alibabacloud.com/product/maxcompute.

[37] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F Wenisch. 2021. Parslo: A Gradient Descent-based Approach for Near-optimal Partial SLO Allotment in Microservices. In *Proceedings of SoCC*. 442–457.

[38] Hiep Nguyen, Zhiming Shen, et al. 2013. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proceedings of ICAC*.

[39] Yaniv Ovadia, Emily Fertig, et al. 2019. Can you trust your model's uncertainty? Evaluating predictive uncertainty under dataset shift. In *Proceedings of NeurIPS*.

[40] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: A Graph Neural Network based Proactive Resource Allocation Framework for SLO-Oriented Microservices. In *Proceedings of ACM CoNext*.

[41] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *Proceedings of OSDI*.

[42] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. 2009. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of ICAC*.

[43] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *Proceedings of SoCC*.

[44] Krzysztof Rzadca, Pawel Findeisen, et al. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of EuroSys*.

[45] Zhiming Shen, Sethuraman Subbiah, et al. 2011. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of SoCC*.

[46] Konstantin Shvachko, Hairong Kuang, et al. 2010. The hadoop distributed file system. In *Proceedings of MSST*.

[47] Nitish Srivastava, Geoffrey Hinton, et al. 2014. Dropout: a simple way to prevent neural networks from overfitting. *In JMLR* (2014).

[48] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of NeurIPS*.

[49] Swarm. [n.d.]. https://docs.docker.com/swarm/.

[50] Dustin Tran, Mike Dusenberry, et al. 2019. Bayesian layers: A module for neural network uncertainty. In *Proceedings of NeurIPS*.

[51] Yijun Xiao and William Yang Wang. 2019. Quantifying uncertainties in natural language processing tasks. In *Proceedings of AAAI*.

[52] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of ICML*.

[53] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic Scaling for Microservices with an Online Learning Approach. In *Proceedings of ICWS*.

[54] Jerrold H Zar. 2005. Spearman rank correlation. *Encyclopedia of biostatistics* (2005).

[55] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices. In *Proceedings of ASPLOS*.

[56] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of EuroSys*.